A Parallelizing Compiler for the Functional Programming Language EVE

Diplomarbeit

zur Erlangung des akademischen Grades
“Diplom-Ingenieur”
in der Studienrichtung
Informatik

eingereicht von

Hans Wolfgang Loidl

Angefertigt am
Institut für Mathematik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eingereicht bei
o.Univ.-Prof. Dr. Bruno Buchberger

Mai 1992

Abstract

This thesis describes the design and implementation of an automatically parallelizing compiler for the purely functional programming language EVE. The target language of the compilation process is the assembler code of the abstract dataflow machine ADAM that has been implemented in the frame of another thesis. In this compilation process dataflow graphs are used as an intermediate language. These dataflow graphs must then be translated into a controlflow program with explicit statements for creating and synchronizing parallel threads of control. A prototype version of the EVE compiler is running.

Keywords: *Parallel Computation*, *Compilation*, *Automatic Parallelization*, *Functional Programming*, *Dataflow Architectures*

Contents

List of Tables

List of Figures

Acknowledgements

Most of all, I want to thank Prof. B. Buchberger, whose enthusiasm in his research inspired me to work in the area of symbolic computation. And I want to thank him and all the other people at RISC who helped in building such a great environment here in Hagenberg: It's really fun to work here.

I want to thank Wolfgang Schreiner very cordially for guiding me in the whole process of my diploma thesis and for his help he has given me so often not only in the frame of my thesis. His outstanding work he has done in his diploma thesis, which was the foundation I was able to base my work on, was itself the best start I could get and was a guideline for my whole work. And I want to express my thanks to all the people in the Parallel Computation Lab at RISC whose work and whose experiences were and are extremely valuable. Especially, I want to thank Kurt Siegl, Peter Zettler, Klaus Kusche, Andi Neubacher and Dr. Hoon Hong, who as the head of our lab gave me very useful advices for my thesis. I hope that the collaboration of us all will be as productive in the future as it was in the past.

Among all the colleagues at our institute I am very grateful to those who always took care about our working environment and helped to facilitate the work of us all. Especially, I want to thank Andi Neubacher and Peter Pregler for providing means to increase the motivation in all phases of my work.

Furthermore I want to acknowledge the work of other scientists all over the world that was helpful for me in writing my thesis. Especially, I am grateful to K. Traub, who presents his work in such a way that other people can get the maximal usage out of it.

Finally, I want to thank all colleagues at RISC for creating such a warm atmosphere that is the ideal environment for really productive work.

Chapter 1

Introduction

In the development of computer science during the last decades one can see two major trends that compete with each other:

- One trend tries to increase the **abstraction** in programming,
- another trend tries to increase the **efficiency** in programming.

The overall development of modern *imperative programming languages* mainly deals with the first trend. The reason for this is the fact that all these programming languages are derived from the structure of a von-Neumann processor and are therefore nothing but abstractions from the assembler language. From this one can see that in imperative languages efficiency has always been the most important aspect and only those abstractions were performed that did not cause much problems with respect to efficiency. However, when writing really large program packages it shows that the abstractions that have been introduced are not really sufficient to create such big packages in a convenient way. This has led to the development of the concept of object oriented programming.

Contrary to this kind of programming languages *functional programming languages* deal with the second trend as they are already on a quite high level of abstraction. They are not abstractions of some machine oriented language but they are an attempt to use at least a part of the usual mathematical language as a programming language. Therefore, the main problem of these languages has always been their lack of efficiency compared to imperative programming languages. However, in the last years big progress has been made in implementing such languages efficiently. Already in [Kranz *et al.*, 1986] a compiler for Scheme has been presented that creates code that is in some cases as efficient as C code. Currently, the implementations of Standard ML (in the New Jersey version) (see [Appel and MacQueen, 1987], [Appel, 1989]) and of Scheme (see [Kranz, 1988]) are some of the most efficient implementations of such languages.

One important possibility to increase the efficiency of programs is **parallel computation**. The idea of this concept is to divide a program in several parts that can be computed independently and therefore on different processors. The problem that *imperative languages* have with parallelism is the fact that they still reflect the idea of having just one thread of control. Therefore, the language model has to be extended to incorporate the idea of several independent processes that communicate with each other. Some of the models that extend imperative languages with a concept of parallelism are:

- The model of *Communicating Sequential Processes (CSP)* (see [Hoare, 1985]),
- a model for an entire *Parallel Program Design* (see [Chandy and Misra, 1988]),
- a *Calculus of Communicating Systems* (see [Milner, 1980]), which presents a mathematical model for parallel computation,
- the *Actor Model* (see [Agha, 1986]),
- the *Linda* model (see [Carriero and Gelernter, 1990]).

Many of the current parallel imperative programming languages are based on the concept of CSP. The most prominent of these languages are:

- *Parallel C*. As nowadays C is one of the most used programming languages it has also been one of the first languages that was extended to incorporate parallel features. Therefore, quite many parallel extensions to C have been implemented. Some of these implementations of parallel C are described in [Gehani and Roome, 1986], [Inmos, 1989] and [Poplet, 1987].
- *Occam*. This language has been designed especially for parallelism. It is a Pascal-like language without recursion that is only implemented on transputers (see [Inmos, 1988]).
- *Modula-P*. This language in some sense extends Occam with the sequential constructs that are known from Modula-2 e.g. for separating a program into various modules (see [Vollmer, 1990]).

The problem for the programmer with this kind of languages is the fact that *parallelism adds an entire new level of difficulty to programming*. These new problems that arise in parallel programming are:

- One has to split the program in processes of appropriate size and one has to decide in which order to execute these processes determining which computations can be done in parallel. For getting efficient programs, the size of the different processes (the *granularity*) must be appropriate. If it is too large, probably not all processors are used all the time; if it is too small the overhead of synchronization and communication might become too big.
- One has to specify on which processor each of these processes shall be executed. This problem is referred to as the *mapping* problem. This mapping can be fixed at compile time or it can be deferred to the runtime. The idea behind the latter strategy is to avoid a situation where some processors have no work whereas others are heavily loaded. However, to distribute the work correctly, information about the load of all processors is necessary. But getting this information includes some overhead. Hence, the right strategy is always a trade-off between these two extremes. This distribution of work among the processors is called *load balancing*.

- One has to handle the communication between the processes in a correct way. Especially, it has to be guaranteed that never a situation occurs where in a set of processes all processes wait for a message from another process of this set. As all processes are blocked in this case this is called a *deadlock*. On some architectures it is even necessary to explicitly program the routing of messages between different processors.

As some of the above specifications depend on the number of processors that are used or on the topology of the connections between these processors the resulting programs are generally not portable between different architectures. Therefore, one loses a big part of the abstraction one has achieved with sequential imperative languages.

Another approach to getting parallel imperative programs is to automatically parallelize imperative programs. The idea behind this approach is that with an automatically parallelizing compiler it should be possible to generate code that can be executed in parallel. By this, it would be possible to increase the efficiency of already existing imperative programs. This is especially interesting as with this method it should be possible to execute existing programs in parallel without having to change these programs at all. However, the big problem with this approach is the fact that in the imperative program almost all of the parallelism that might have been contained in the given problem has been destroyed in order to get one single thread of control. Therefore, it is an extremely difficult task to regain this parallelism by an automatically parallelizing compiler. Hence, today it usually can not be avoided to help the automatically parallelizing compiler by providing some additional information about the execution of the program.

Contrary to these imperative languages, *functional languages* do not define a sequential thread of control. Moreover, they define a partial ordering on the operations that have to be executed in order to get a result. As all operations that do not depend on each other (i.e. that are incomparable in that partial ordering) can be executed in parallel, we can say that all functional languages contain some *implicit* parallelism. This presence of implicit parallelism is the main reason why functional languages are well suited for parallelism. Of course, it is also possible to extend the language with constructs for *explicit* parallelism[1]. By using such an extended language, some tasks (like splitting the program in parallel processes of the appropriate size) that could also be done by a compiler are left to the programmer in order to get more efficient programs. A more detailed discussion of parallel functional programming will be given in Chapter 2.

At the RISC institute the interest for parallel computation comes from many very time consuming algorithms especially from computer algebra. Some of these algorithms whose parallelization are especially interesting for us are the Gröbner Bases algorithm by B. Buchberger (see [Buchberger, 1985]) and the algorithm for Cylindric Algebraical Decomposition by G. Collins (see [Collins, 1975]). In this area of symbolic computation, functional languages are very interesting as they allow to express a program in a similar way as in the usual mathematical notation. In another thesis ([Siegl, 1990]) it has already been shown that by using a declarative language it is possible to get a significant speed-up for the Gröbner Bases algorithm. Therefore, at the RISC institute the ADAM & EVE project has been started in 1989. The main goal of this project was to study the behaviour of programs in the functional programming language EVE that have been compiled to the assembler code

[1] Such an approach for explicit parallelism in a functional language will be shown in Section 2.2.

of the abstract dataflow machine ADAM[2]. This approach to parallelism has the following advantages:

- The EVE language is well suited for solving problems in the area of symbolic computation[3].
- The EVE language is implicitly parallel. The programmer does not have to include any statements concerning parallelism. However, in order to get efficient parallel programs one has to take the data dependencies between the various objects in an EVE program into account. See Section 5.2 for a comparison of the parallel behaviour of two similar programs with different data dependencies.
- As the ADAM machine simulates dataflow behaviour on a controlflow machine, conventional hardware can be used to execute EVE programs.

This thesis forms the second part in the ADAM & EVE project. The first part consisted of the implementation of the ADAM machine on a transputer network and the design of the EVE language. This part is described in [Schreiner, 1990].

The structure of the whole thesis is as follows:

- After the introduction in this chapter, Chapter 2 gives a survey of **parallel functional programming**. This chapter first gives a general introduction to functional programming. Afterwards the introduction of explicit parallelism into a functional language is discussed. Finally, some abstract machines for the implementation of functional languages are presented and it is described how these abstract machines can be parallelized.
- Chapter 3 gives a short description of the internal structure of the **ADAM machine**. As the design and implementation of this machine was the topic of the first part of the ADAM & EVE project here only a summary of the results of this part will be given. Furthermore, this chapter shows the **specification of the EVE language**. In the realization of this project some modifications have been performed on the EVE language. These modifications are emphasized in this chapter[4].
- The main part of the whole diploma thesis is Chapter 4 where the implementation of the **EVE compiler** is discussed in detail. In this chapter the translation of the EVE language into ADAM assembler code is discussed in two steps. In the first step the translation of all EVE constructs into dataflow graphs is shown. In the second step these dataflow graphs are translated into ADAM assembler code.
- Chapter 5 contains a discussion of the **experiments** that have been performed with this compiler. There also some experiences with implicit parallel programming and the translation to code that simulates a dataflow behaviour are presented. Furthermore, some ideas for optimizing the generated code are discussed there.

[2]Actually, the ADAM machine is a hybrid dataflow/controlflow machine. For a detailed discussion of this machine see Chapter 3.

[3]Compared with other current functional languages EVE is rather austere and therefore not as convenient to use as these languages. However, it has all the principle advantages of a functional language.

[4]However, the specification of EVE in this thesis is complete and therefore no other references are needed to understand the details of the language.

- Chapter 6 summarizes the **results** of this thesis but also the results of the whole ADAM & EVE project.
- In Appendix A a **formal description of the syntax analysis** (the first pass in the compiler) is presented in the form of an attribute grammar.
- Appendix B shows a **formal description of the code generation** in the EVE compiler. This description is presented in the form of a denotational semantics. As this way of describing the code generation hides the operational aspect, it should be quite easy to read. Nevertheless, it gives a complete description of the process.
- Appendix C gives a summary of the **ADAM instruction set** and it discusses syntax and semantics of each ADAM instruction.
- Appendix D shows some **EVE programs** with which benchmarks have been made. Furthermore, these programs are referred to in Section 3.2 as typical examples for EVE programs.
- Finally, Appendix E shows the **definition modules** of all parts of the EVE compiler. As these definition modules are documented, this also gives a description of each procedure that has been used in the Appendices A and B.

Chapter 2

Parallel Functional Programming

This chapter gives a general survey of parallel functional programming. In the first section an introduction to functional programming will be given. This part will be quite general without discussing details of the EVE language. However, at some points EVE syntax will be used for some expressions (these parts are written in `typewriter` style). Although the concrete syntax of EVE will be discussed in Section 3.2 the semantics of the expressions that are used in this chapter should be obvious from the context. Furthermore, it is discussed in this chapter how explicit and how implicit parallelism can be used in a functional language. For that some abstract machines for implementing functional languages are presented and it is shown how parallelism can be used in these abstract machines.

2.1 Introduction to Functional Programming

In this section we give a short introduction to functional programming in general. Due to the lack of space only the most important features of functional programming languages will be discussed here. For a detailed introduction the interested reader may refer to [Bird and Wadler, 1988] and [Field and Harrison, 1988]. Both books give an excellent introduction to the various aspects of functional programming without requiring any previous knowledge in that field.

During the last years many functional languages have been developed. The most prominent functional languages today are the following:

- **Haskell** (see [Hudak *et al.*, 1991]) is the most modern functional language. It has been designed by a committee that represents the modern school of functional programming. It is an attempt to create a standard functional language and by that to base the research in this area on a common language. This approach is motivated by the fact that most of the other functional languages differ just in syntactic details but do not include important semantic differences. The innovative new feature of Haskell is the handling of classes that form a hierarchical structure of types with the corresponding functions on it. For an introduction to Haskell see [Hudak and Fasel, 1992]. A definition of the static semantics of Haskell can be found in [Peyton Jones and Wadler, 1991].
- **SASL** (see [Turner, 1976]) was one of the first languages that included the concept of non-strictness. It also already included some kind od pattern matching. Note that SASL was the predecessor of Miranda.

- **Miranda**[1] (see [Turner, 1985]) is one of the most popular functional languages that has a syntax that is in many areas very similar to that of Haskell.
- **Standard ML (SML)** (see [Milner, 1984], [Wikström, 1987], [Harper, 1990]) has as its main advantage a rich type system. Furthermore, this language has also become very popular since there exist very efficient compilers for many different machines. However, one major difference to the above languages is the fact that SML has a strict semantics of user defined functions. For a discussion of the notion of strictness see Section 2.1.1.
- **Lazy ML (LML)** (see [Johnsson, 1987], [Augustsson, 1987]) is the version of ML with a non-strict semantics. The compiler for this language that is described in the above papers is based on the G-machine, which is one of the most important abstract machines for graph reduction (see Section 2.3.2).
- **Hope** (see [Burstall *et al.*, 1980]) is used in [Field and Harrison, 1988] to describe functional programming in general and the implementation of functional programming languages.
- **Ponder** (see [Fairbairn, 1985]).
- **Orwell** (see [Wadler, 1985]).
- **Id** (see [Nikhil, 1988]) is a functional programming language augmented with a parallel data-structuring mechanism called I-structures. It is especially interesting for us as it was designed for a parallel implementation on a dataflow machine (see also [Traub, 1986]).
- **FL** (see [Backus *et al.*, 1990]) has developed out of the ideas that were presented in the famous talk of J. Backus ([Backus, 1978]) that marks the actual start of modern functional programming. However, its syntax notably differs from the syntax in most other functional languages.

The language that will be used in the language specific parts of this chapter is EVE (these parts are written in `typewriter` style). However, as we will concentrate on general concepts of functional programming in this chapter, the results can be also applied to most of the other functional languages mentioned above.

The main characteristic of functional languages is *referential transparency*. This means that the result of the evaluation of an expression only depends on the values of the free variables in that expression. Or in other words: The evaluation of an expression does not perform any side-effect. For functions that means that two calls of a function with the same parameters and same values of global variables always yield the same result.

One consequence of this characteristic is the fact that there are no variables in the sense of variables in imperative languages. There a variable is considered as a box where one can put values into and take values out of. Contrary to this *box model*, in functional languages a variable *stands* for a certain value. Therefore, one can say that the variable is *bound* to this value. This is also sometimes called the *single assignment* rule of functional languages (although this is no assignment in the usual sense).

[1] Miranda is a trademark of Research Software Limited.

Another consequence of referential transparency is the fact that for any expression it is possible to evaluate its subexpressions in any order, especially they can be evaluated in parallel. The theoretical justification for this is given by the *Church-Rosser Property* (an exact formulation and a proof of this property can be found in [Barendregt, 1984]). This property states that for the evaluation of an expression to its result any evaluation strategy may be chosen. This property is the basis for all parallel implementations of functional programming languages.

To explain this property with a concrete example let us look at the following expression:

$$square\ (1+2)$$

where the function *square* is defined as follows:

$$square\ x \;=\; x * x.$$

One possible evaluation strategy of this expression is first to evaluate the argument $(1+2)$ and then to use the definition of *square* to get the final result of the expression:

$$\begin{array}{rcl} square\ (1+2) & \rightarrow & square\ 3 \\ & \rightarrow & 3*3 \\ & \rightarrow & 9. \end{array}$$

However, another possible evaluation strategy is to insert the unevaluated argument $(1+2)$ into the definition of *square* and to evaluate the argument afterwards:

$$\begin{array}{rcl} square\ (1+2) & \rightarrow & (1+2)*(1+2) \\ & \rightarrow & 3*3 \\ & \rightarrow & 9. \end{array}$$

From this evaluation sequence one can see how subexpressions can be evaluated independent of each other.

One interesting feature about these different kinds of evaluation strategies is that they express different degrees of expressive power of the underlying language. Therefore, the choice of the evaluation strategy depends on the definition of the semantics of the underlying language. This is discussed in more detail in Section 2.1.1.

It is beyond the scope of this thesis to give a detailed introduction to all aspects of functional programming. Therefore, we give, after having explained the basic idea of functional programming, just a short sketch of the most important features of functional programming languages:

- Functional programming languages offer a very convenient way of defining *compound data structures* and of handling such data structures. Typical classes of types that can be defined are lists and tuples.

- In most functional programming languages it is allowed to use patterns instead of simple variables at the left hand side of a function definition. When such the function is called the structure of this argument is compared wih the pattern in the function definition. Only when they match the right hand side is evaluated and taken as the result of the function. In the evaluation of the right hand side the variables in the pattern are bound to the appropriate components of the argument. This whole mechanism is called *pattern matching*. With this construct it is possible to develop an *equational style* of defining functions that has a strong resemblance to the usual mathematical notation.

- A very powerful construct of functional languages are *higher order functions*. These are functions that take other functions as arguments or yield a function as a result. The power of such functions lies in their flexibility. For example in most functional languages there is a function `map` that applies another function `f` that is an argument of `map` on each element of a list `l` that is the second argument of `map`. By using this function `map` with different functions `f` one can solve many different problems.

- Many functional languages allow a special way of defining lists that bears a strong resemblance to the way how sets are usually defined in mathematical notation (ZF-notation). This construct is called a *list comprehension*. However, as some functional languages also have the builtin type of an array available, in these languages this construct is enhanced to the construct of an *array comprehension*. Note that especially array comprehensions contain much parallelism as one can describe quite complex data dependencies with this construct. For a detailed discussion of this so called *wave front parallelism* in array comprehensions see [Hudak, 1986a].

- Another possibility that offer most functional languages is to call a function with fewer parameters than it was defined. The result of this *currying* is a function in the remaining arguments.

- User defined operations can usually be defined as *prefix* or as *infix* operations. Furthermore, prefix operations can be transformed into infix operations and vice versa. This flexibility in the usage of user defined functions increases the readability of functional programs very much.

For a detailed explanation of all these functional language constructs see [Bird and Wadler, 1988].

2.1.1 Strictness

One of the most important features of a functional language is its strictness property. Formally, strictness can be defined as follows:

Definition 1 (Strictness) *An n-ary function f ist* strict *in its i-th argument iff*

$$\forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \; : \; f(x_1, \ldots, x_{i-1}, \perp, x_{i+1}, \ldots, x_n) \; = \; \perp$$

where $\perp$ stands for non-termination or an error.

If a function is strict in its i-th argument it is safe to evaluate that argument before the function itself is called. The reason for that is that if the evaluation of the argument does not terminate (and thus yields $\perp$) the above definition tells us that also the whole function application does not terminate. On the other side if a function is non-strict in its i-th argument the function might return a result even when the evaluation of the i-th argument does not terminate. To make this clearer let us look at the usual if-then-else construct:

$$\texttt{IF}\ E_1\ \texttt{THEN}\ E_2\ \texttt{ELSE}\ E_3.$$

This construct is obviously strict in the first argument (E_1) as it can not return a result when the evaluation of the boolean expression E_1 does not terminate. However, it is non-strict in the other two arguments (E_2 and E_3) as only one of these arguments will actually be evaluated.

One can generalize this definition of strictness to data structures when we define a data structure to be strict iff its constructor function is strict. The consequence of a non-strict data structure is that the data structure can be used although not all of the components of this data structure have been evaluated. Therefore, in any implementation of non-strict data structures it must be guaranteed that a value is written into a field of a data structure before this field is actually read.

The strictness property of a language determines the expressive power of the language. The reason for that is the fact that the evaluation strategy is determined by the strictness property of the language. If the language is strict (i.e. if all user defined functions are strict in all arguments) then an *eager* evaluation strategy can be chosen that evaluates the arguments of the function before the function itself is called (therefore this evaluation strategy is also often called *call-by-value* strategy). Note that all imperative languages are strict and therefore use this evaluation strategy. Some examples for strict functional programming languages are SML and Hope. However, when the language is non-strict usually a *lazy* evaluation strategy is used, that evaluates the argument of the function as soon as it is needed inside the function but usually not already when the function is called (therefore this evaluation strategy is also often called *call-by-need* strategy). Note that this is not the only possible evaluation strategy that fulfills non-strictness. Other possible evaluation strategies would be a "parallel non-strict" strategy that evaluates the argument in parallel with the function. If this strategy is fair a non-termination of the evaluation of the argument does not necessarily mean that the evaluation of the function does not terminate either. Another evaluation strategy that conforms to a non-strict semantics but that is different from the lazy evaluation strategy is the *lenient* evaluation strategy that is described in [Traub, 1991]. Examples for functional languages with a non-strict semantics are Haskell, Miranda, LML, SASL, Ponder, Orwell.

The increase in expressive power in non-strict languages has two consequences:

- In non-strict languages *circular data structures* can be defined. One example for such a data structure is an (conceptually) infinite list of ones:

  `ones : IntList == 1 :: ones`[2].

---

[2] The type `IntList` is defined as follows:

```
IntList == LIST OF INT.
```

Such a list is called a circular list. As this list can be represented by only one cell in which the tail points to the cell itself, this is a very efficient representation of a list.

- In non-strict languages *circular data dependencies* can be expressed. One example for that is the following definition of a list of all Fibonacci numbers:

```
AddLists[l1,l2] : IntList[IntList,IntList] ==
   IF NULL[l1]
     THEN l2 ELSE
   IF NULL[l2]
     THEN l1
     ELSE {l1hd : INT == HEAD[l1];
           l1tl : IntList == TAIL[l1];
           l2hd : INT == HEAD[l2];
           l2tl : IntList == TAIL[l2];
          }
          l1hd+l2hd :: AddLists[l1tl,l2tl];

MAIN : IntList == 1 :: 1 :: AddLists[MAIN,TAIL[MAIN]];
```

  The idea of this definition is to get the list of all Fibonacci numbers starting from the third element by componentwise adding the list of all Fibonacci numbers with the list of all Fibonacci numbers starting with the second element. Hence, the third element of the list `MAIN` is the sum of the first and the second element. Therefore, in the list `MAIN` there is a circular data dependency as the i-th element of the list depends on the $(i-1)$-st and on the $(i-2)$-nd elements. However, there is no circular data structure created as it was the case in the list `ones` above. As each element of the list `fibs` is only evaluated once and can then be used for the evaluation of succeeding list elements this is already a definition where the computation of the i-th element takes linear time (contrary to the straightforward definition that takes exponential time). From this one can see that circular definitions are also very time efficient. For a detailed discussion of the efficiency of such circular definitions see [Loidl, 1991].

However, a non-strict language has not only more expressive power but it also contains more inherent parallelism than a strict language:

- In a non-strict language the function itself can be evaluated in parallel with its arguments. When the function returns a value even before all arguments have been finished the result need not explicitly synchronized with the unevaluated arguments.

- In a non-strict language a data-structure may be used before all of its components have been evaluated. This fact allows a very convenient way to express the so called *producer-consumer parallelism*, where one process (the producer) creates a big data structure and another process (the consumer) performs some operations on the components of this data structure. Due to the non-strict semantics of data-structures, both processes may be executed in parallel when they are connected with an ordinary function composition. To make this clearer, let us look at the following example:

```
producer[n] : IntList[INT] ==  n :: producer[n+1]
```

```
consumer[l] : IntList[IntList] ==
   { hd : INT == HEAD[l];
     tl : IntList == TAIL[l]
   }
   square[hd] :: consumer[tl];

squares : IntList == consumer[producer[1]];
```

In this example a list of the squares of all natural numbers is computed by executing the functions `producer` and `consumer` in parallel. As soon as `producer` has created a list element, this element can be taken by `consumer` and squared. Note that this definition of `squares` is equivalent to the following definition (the function `From[1]` creates a list of all integers starting from `1`):

```
squares : IntList == map[square,From[1]]
```

Even the runtime behaviour is similar to the above definition as `map` is a non-strict function that can be evaluated in parallel with `From[1]`. However, the first definition makes the producer-consumer structure more explicit.

Because of these advantages of non-strict languages compared to strict languages, we will only discussed non-strict languages in the further chapters.

2.2 Explicit Parallelism in Functional Languages

As it has already been mentioned in Chapter 1, functional languages are well suited to parallel computation. The main reason for that is the fact that these languages contain much inherent parallelism that can be exploited by a compiler. However, it is also possible to add explicit parallelism to these languages. In this section we will discuss this approach to parallelism.

If one wants to enhance a programming language by explicit parallel constructs one has to implement the following operations:

- The *creation* of a new thread of control,
- the *synchronization* of several threads of control and
- the *communication* between different threads of control.

In [Reppy, 1991] this has been done by extending Standard ML yielding an explicitly parallel functional language that is called **CML**. In CML the creation of a new thread of control is performed by a call to the function `spawn`. This function takes another function as argument that will be evaluated in the new thread of control. The return value of the `spawn` function is an identifier of the new thread. The thread terminates when the evaluation of the function has been finished or when a special function `exit` is called. For the communication between the different threads *channels* are used. Via these channels synchronous message passing is possible by using the functions `send` for sending a message on a channel and `accept` for reading a message from a channel. However, in [Reppy, 1991] it is also shown how a dataflow like style of programming can be used by implementing streams in this model. Furthermore, it is shown how *futures* as they are introduced in [Halstead, Jr., 1985] can be simulated in this model. A new data type *events* is added to the language that represents synchronous operations. To make the handling of parallelism in that model easier some functions are defined on this new data type. The actual simulation of parallelism is performed by passing continuations.

Note that with this way of introducing parallelism to a functional language referential transparency is lost. However, with SML this is not so serious as it is an impure functional language in the sense that it contains instructions that perform side-effects.

A similar approach has been taken in [Cooper and Morrisett, 1990]. The functions for the creation and synchronization of threads are similar to those in the above paper. However, the later thread package also contains an own data type for mutual exclusion locks. Furthermore, in [Cooper and Morrisett, 1990] the implementation of this package under the operating system Mach is discussed in detail.

However, both papers show that the problems in parallel programming that arise when using explicit parallelism in a functional programming language are essentially the same as the problems with explicit parallelism in an imperative programming language. On the other hand, it is possible to implement with this approach quite efficient parallel algorithms as one has direct influence on the parallel behaviour of the program but one can not exploit the implicit parallelism of the functional programming language.

2.3 Parallel Implementation of Functional Languages

In order to extract the implicit parallelism out of a functional program one needs an appropriate abstract machine into which the functional program is translated. However, for the sequential implementation of functional languages several abstract machines have been introduced in the literature. These machines vary in their degrees of abstraction from the physical machine model. But even more important, they are based on different models of computation that can be distinguished by the way how the computation process is driven.

For a parallel implementation of functional languages it is now essential how parallelism can be expressed in these different models of computation. Luckily, most of these models are on a quite high level of abstraction such that they do not reflect the sequential computation model of a von-Neumann processor any more. The reason for that high level of abstraction is the fact that the functional languages themselves are on a very high level of abstraction.

The most important kinds of **abstract machines for the compilation of functional programming languages** that have been introduced so far are:

- The **lazy SECD machine**,
- **graph reduction machines**,
- **dataflow machines**.

One of the best books dealing with the sequential implementation of functional programming languages is [Peyton Jones, 1987]. In this book a graph reduction machine is used as underlying abstract machine. In one chapter an outlook to parallel graph reduction is given. The new book [Peyton Jones and Lester, 1992] gives a practical approach to understanding implementations of non-strict functional languages using lazy graph reduction. Concerning parallelism, the parallel G-machine is presented in this book. Furthermore, [Field and Harrison, 1988] also discusses functional languages and their sequential implementation. However, in this book all of the three abstract machine models that are discussed here are covered. For a general survey of parallel functional programming including presentations of sequential and parallel implementation techniques and of parallel functional programming techniques see [Kelly, 1989]. Another discussion of the parallel implementation of functional programming languages that is mainly based on parallel graph reduction machines can be found in [Peyton Jones, 1989]. Furthermore, [Szymanski, 1991] discusses various approaches to parallel functional programming, each of which is presented by another author.

In the following sections we will now study the parallelism that is contained in each of these abstract machines. For that we will for each abstract machine first present the model of computation upon which it is based. Then it will be discussed how much parallelism can be exploited in this model. Furthermore, for each machine examples of sequential and parallel implementations will be given.

2.3.1 The Lazy SECD Machine

The SECD machine is a stack based machine that has been introduced in [Landin, 1964]. However, the classical SECD machine can only be used together with eager evaluation i.e. the arguments are evaluated before they are passed to a function (call-by-value). In order to get a machine that can handle lazy evaluation i.e. the arguments are only evaluated when their results are really needed (call-by-need) some changes in the machine have to be performed. The main change is the introduction of a suspension consisting of an expression and an environment. Such a suspension represents an unevaluated argument of a function.

One of the first extensions of the SECD machine for lazy evaluation is given in [Burge, 1975]. Such a lazy SECD machine has been used in an early implementation of SASL (see [Turner, 1975]). Another variant of the lazy SECD machine is the FLFM machine (see [Takeichi, 1986]) upon which an implementation of the language RUFL is based (see [Wentworth, 1989a]). As this is a rather conventional model of computation the whole evaluation process is driven by an explicit controlflow. Therefore, this model can be called a *control-driven* model.

As the SECD machine is based on the principle of a single controlflow it is quite badly suited for a parallel implementation. Moreover, even in the sequential case the other models have proven to be as efficient as the lazy SECD machine. But on the other hand it is easier to compile a functional program to the code of a graph reduction or a dataflow machine as these two machines are on a higher level of abstraction. Therefore, only these two models will be discussed with respect to their parallelism.

2.3.2 Graph Reduction Machines

The idea in this model is to represent a functional program as a graph and to execute the program by applying transformations ('rewriting') on this graph. This idea is shown in Figure 2.1 where the expression $(2 + 3) * (4 - 1)$ is evaluated by performing rewrite operations on its graphical representation. Note that it is in general not necessary to evaluate the arguments before the rewriting of the function can be performed. In Figure 2.1 the arguments are evaluated first because the top-level function $(*)$ is strict. If it were non-strict the values of the arguments would not be needed and the rewriting of the whole function could be performed immediately. Only when the function is strict the evaluation of the arguments has to be demanded before the rewriting of the function itself can be done. Due to this behaviour this model is also called a *demand-driven* model. It is especially useful for implementing lazy evaluation.

In general, these rewrite rules can be a fixed set of rules as in the case of the SKI-reduction machine that is described in [Turner, 1979] or the rules might be derived as so called 'super-combinators' from the given program itself[3]. This method has first been used in [Hughes, 1982].

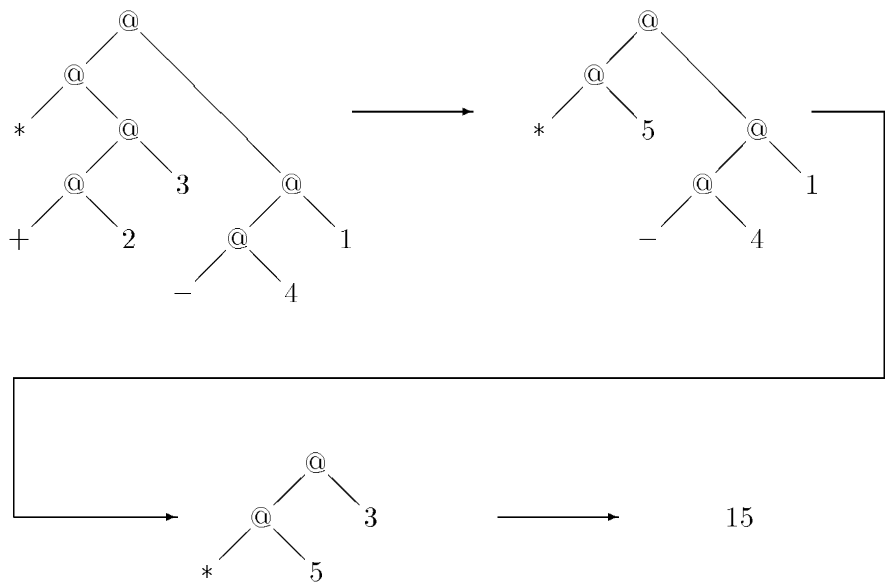


Figure 2.1: Sequential Graph Reduction of the Expression $(2 + 3) * (4 - 1)$

One machine for the execution of super-combinators is the Three Instruction Machine (TIM) ([Fairbairn and Wray, 1987]) which is used for the implementation of the functional language Ponder. This super-combinator approach has also been taken in the G-machine (see [Johnsson, 1987], [Augustsson, 1987]) and this has led to one of the most efficient implementations of a lazy functional language (LML) up to now. The characteristic novelty in the G-machine is the fact that each function is translated into a piece of code that simulates the process of graph reduction. Therefore, when a function is called this code sequence will be executed instead of performing an interpretive graph reduction. An improved version of the G-machine, namely the spinless, tagless G-machine is currently used in an implementation of Haskell at

[3] For an explanation of the notion of 'super-combinator' and a further discussion of the differences between these two approaches see Section 4.3.

the University of Glasgow. In [Peyton Jones, 1991] this spineless, tagless G-machine is presented. Especially, the differences to the original machine and the mapping of the machine onto hardware by using C as a portable assembler are discussed in detail there.

Now let us consider how a **parallel implementation** of this machine could be realized. The obvious idea to introduce parallelism into this model is to simultaneously rewrite independent subgraphs. Due to referential transparency the result with this reduction strategy will be the same as the result with a sequential reduction strategy.

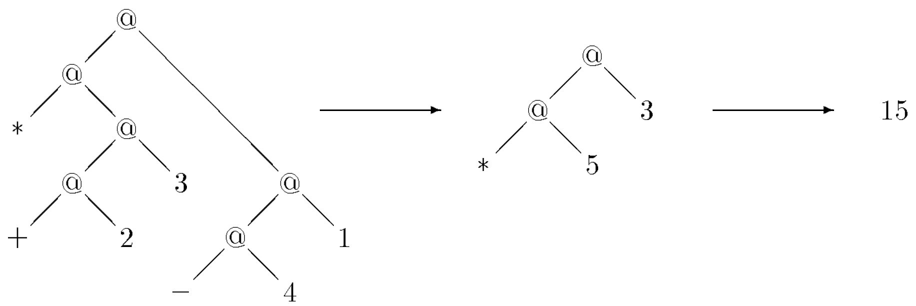


Figure 2.2: Parallel Graph Reduction of the Expression $(2 + 3) * (4 - 1)$

Figure 2.2 shows the parallel reduction process for the same expression as in Figure 2.1. Note that in this case one step less is needed as the subexpressions $(2 + 3)$ and $(4 - 1)$ can be evaluated in parallel.

We will now discuss the principle structure of an algorithm for parallel programmed graph reduction. For that we assume an architecture consisting of the following components:

- One graph that represents the actual state of the program,
- a memory containing code for the evaluation of every function in the program (as it has been described for the G-machine this code simulates the reduction of a graph without performing operations on the graph itself),
- a task pool containing all those subgraphs that have to be reduced,
- several processors.

Algorithm for parallel programmed graph reduction:

1. The root of the graph is put into the task pool.
2. One of the idle processors takes a task out of the task pool.
3. When a processor comes in the evaluation of his task to a call of a function $(f\ E_1\ \ldots E_n)$ he determines which expressions are needed for the evaluation of f[4]. The expressions

[4] This can be done by performing a strictness analysis on the function f at compile time. As a result this analysis yields the information which arguments of the function are in any case needed for the evaluation of f itself.

that are needed for the evaluation of f are put into the task pool and the evaluation of the function application is blocked.

4. Other processors take these expressions out of the task pool and evaluate them.

5. As soon as all necessary expressions have been evaluated the application of f can be evaluated[5].

Although the above algorithm is very general, it shows the principle handling of parallelism. Especially, it shows that the creation of parallelism only occurs in function applications. Furthermore, strictness analysis is used to determine which expressions can be evaluated in parallel. To be more exact, strictness analysis is used to create parallelism[6]. As only those expressions are evaluated in parallel that are definitely needed for the evaluation of the function this model is based on the idea of *conservative parallelism*.

Some machines that implement this model of parallel graph reduction are:

- The **ALICE** ([Cripps *et al.*, 1987]) machine was the first hardware implementation of a parallel graph reduction machine. It is based on transputers. The programming language for this machine is Hope.

- The **Flagship** machine ([Watson *et al.*, 1988]) is the successor of the ALICE machine. This machine contains special hardware support for the transmission of whole subgraphs between different processors.

- The **GRIP** (Graph Reduction in Parallel) ([Peyton Jones *et al.*, 1987]) machine is a shared-memory multiprocessor designed for efficient parallel evaluation of functional languages, using compiled graph reduction.

- The **PAM** (Parallel Abstract Machine) ([Loogen, 1990]) is another abstract machine for programmed parallel graph reduction that has been implemented on a network of transputers.

- In [Goldberg, 1988] the implementation of the abstract graph reduction machine **Alfalfa** on two different hardware architectures (one shared-memory machine and one distributed memory machine) is presented. However, Alfalfa is actually a hybrid of a graph reduction machine and of a conventional stack oriented machine. It is programmed in the functional programming language ALFL (see [Hudak, 1984]), which bears a strong resemblance to SASL.

- Furthermore, several parallel extensions of the G-machine have been presented in the last years e.g. in [George, 1989], [Burn, 1988] and [Augustsson and Johnsson, 1989].

2.3.3 Dataflow Machines

This model is based on the idea to view the data as the dynamic entities, which flow between a collection of operations, which are the static transformers that manipulate this data. The

[5] In order to notify the function f about the availability of all necessary expressions, each of these expression carries a reference to this application and sends its result to this reference when it has been evaluated.

[6] Note that in the dataflow model it is the other way round.

collection of operations forms a directed acyclic graph with the free variables as its input arcs and the result of the computation as its output arc. This graph is then called a *dataflow graph*. Note that in this model the computation is driven by the availability of data (*data-driven* model). Therefore, this model is especially appropriate for implementing eager evaluation. However, e.g. in [Field and Harrison, 1988][Chapter 14] it is shown how a demand-driven dataflow machine can be realized.

To explain this basic idea of the dataflow model we examine the following mathematical expression:

$$(a+b)*(-y) \textbf{ where } \begin{aligned} a &= -x \\ b &= x/y \end{aligned}$$

Figure 2.3 shows the dataflow graph that corresponds to this expression. In this dataflow graph the nodes represent operators and the arcs between the nodes represent the data dependencies between those operators. The input arcs denote the free variables of the expression and the output arc denotes the result of the expression. In the execution of the program data tokens that carry values flow along these arcs. Whenever at an operator tokens are present at each input arc this operator fires i.e. it takes these tokens from the input arcs, performs an operation on their values and it creates an output token with the computed result.

When one wants to compile a functional language to a dataflow graph there are more than only the arithmetical/boolean operations required. For example the call of another function is in the dataflow graph realized by creating a dynamic link from the call of the function to the function definition itself such that the arguments of the function can be passed to its definition along this arc. A list of all necessary operations with a description of each operation can be found in [Field and Harrison, 1988][Chapter 14]. For a more detailed description of the dataflow model see Chapter 3.1.

It should be clear from this description of dataflow machines that this architecture is very appropriate for a **parallel implementation** as it entirely lacks the concept of a controlflow. Moreover, the only restriction to the parallelism in a dataflow machine is given by the data dependencies between different operations (these are the arcs in the dataflow graph). Therefore, a dataflow machine can extract the maximal parallelism out of a given program. However, this usually introduces too much parallelism to be very efficient. Therefore, one of the main trends in the research of dataflow machines goes into the direction of restricting this parallelism to get bigger blocks of code that can then be evaluated in parallel.

One approach to solve this problem is the introduction of *functional quads* as they are described in [Traub, 1991]. This is a model of functional program execution where the whole program is split into a set of sequential threads. These sequential threads are segments of sequential code, where the instructions are ordered at compile time. However, the order in which the sequential threads execute is not fixed at compile time but it is allowed to vary according to the input data. The run-time switching between the threads is directed only by the communication of values between threads. The scheduling of the threads is controlled by tests to determine whether another subexpression has been already evaluated. These tests are realized by checking presence bits. This reminds of the concept of I-structures in the TTDA architecture.

One main idea behind the implementation of non-strict functional languages as it is presented in the above book is to use an evaluation strategy that has not as much expressive power as

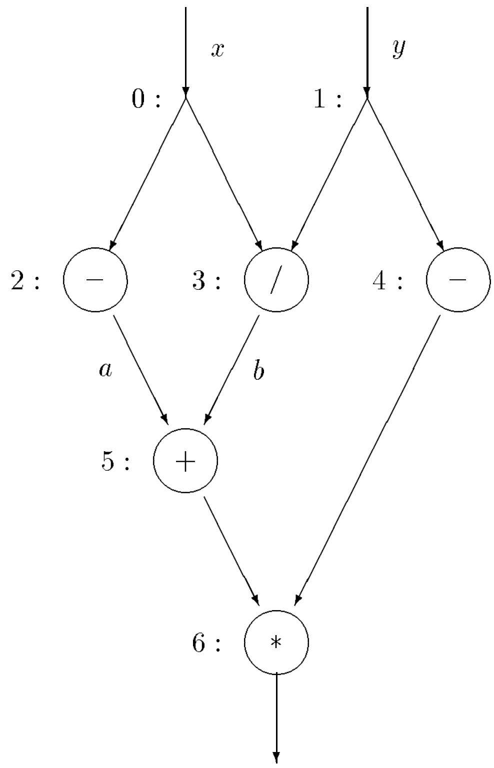


Figure 2.3: A Dataflow Graph

lazy evaluation but still conforms to the non-strict semantics of the underlying functional language. As this evaluation strategy, which is called *lenient evaluation*, does not include as much run-time overhead as the lazy evaluation strategy, this results in more efficient code. With lenient evaluation expressions must be delayed until their input data are available, but not necessarily longer. Hence, it is possible to express with lenient evaluation complicated data dependencies like cyclic data dependencies. But it is not any more possible to use infinte data structures. However, it is still possible to realize infinite data structures with explicit delay and force statements.

Related work has been presented in [Sarkar and Hennessy, 1986], where a technique for partitioning dataflow code into "macro-dataflow" nodes is discussed, and in [Hudak and Goldberg, 1985] such a partitioning is shown for the model of serial combinators.

One machine in which the dataflow model is realized in hardware is the "Manchester Dataflow Computer" that is described in [Ruggherio and Sargeant, 1987]. Another dataflow architecture that was extensively simulated in software is the "TTDA Tagged Token Dataflow Architecture" at the MIT (see [Arvind and Nikhil, 1988]). The architecture that is used there strongly influenced the design of the ADAM machine at RISC (see [Iannucci, 1988]). Moreover, the literature on compiling the functional language Id to the assembler code of the TTDA architecture treats many of the problems that also had to be solved in the implementation of the EVE compiler (see [Traub, 1986]). For the successor architecture, the ETS Explicit Token Store architecture "Monsoon" ([Papadopoulos, 1988]) also a hardware implementation has been realized in a cooperation between the MIT and Motorola (see [Papadopoulos and Culler, 1990]). Another dataflow machine that has been simulated but not built in hardware is "The Stollmann Dataflow Machine" that is presented in [Treleaven, 1990]. However, this machine is mainly intended for applications in database management.

2.4 Using Annotations to Express Parallelism

In the previous sections it has been emphasized that a functional program that should be executed in parallel need not have any statements for explicit parallelism as functional languages are inherently parallel. However, it is a quite difficult task for a compiler to extract parallelism of an appropriate grain size from the program, to schedule and to map these blocks on the available processors in such a way that one really gets an efficient program. Therefore, one might argue that functional programming languages should hide details like scheduling and mapping from the programmer but also to offer the programmer the possibility to have direct influence on these details. The most convenient way to realize this is by using annotations in the functional language to express e.g. the scheduling or the mapping of a functional program.

The most prominent such concept is that of *para-functional programming*. This notion was coined by P. Hudak in [Hudak, 1986b]. Para-functional programming is an extension to the functional programming paradigm that allows to express operational details like scheduling or mapping without losing any of the benefits of functional languages, like the ability to perform equational reasoning on functional programs. Thereby, the functional behaviour of the program is still separated from the operational behaviour. Another advantage of this approach of using annotations is the fact that it can be used with any functional language.

With the annotations that are introduced via the concept of para-functional programming, the operational behaviour of a program can be influenced in two ways:

- One can specify a special *evaluation order* of the program and
- one can specify how to *map* a program to a machine.

Controlling Evaluation Order. The default evaluation order is lazy evaluation. However, this can be changed for any expression in the program by using a *scheduled expression* of the following form:

```
exp sched sexp
```

where `exp`, the body of the scheduled expression, is another expression, `sched` is a keyword and `sexp` is a schedule whose structure is described below. Note that a subexpression in `exp` can be labelled by using the form `lab@exp` for the labelled expression.

A schedule defines a partial order on events, of which for every labelled expression `lab@exp` there are three kinds:

- The *demand* for the evaluation of `exp`, denoted by `Dlab`,
- the *start* of the evaluation of `exp`, denoted by `^lab`,
- the *end* of the evaluation of `exp`, denoted by `lab^`.

Furthermore, the following operations are defined on schedules:

- `s1.s2` denotes the *concatenation* of the schedules `s1` and `s2` (i.e. every event in `s1` precedes every event in `s2`).
- `s1|s2` denotes the *concurrence* of the schedules `s1` and `s2` (i.e. `s1` and `s2` are running in parallel).

With these constructs one can form the following scheduled expressions:

- `(e0 m@e1 n@e2) sched Dm|Dn`. This declares that the demands for the evaluations of the expressions labelled with `m` and `n` will begin in parallel at the point when the entire expression begins evaluation. Thus, this expressions denotes speculative parallelism of the subexpressions `e1` and `e2`.
- `o@(l@e0 m@e1 n@e2) sched l.m.n.o`. This declares a left-to-right call-by-value semantics. Note that in a schedule `lab` is the abbreviation for `Dlab.lab^`. However, this schedule does not prohibit parallelism inside `e0`, `e1` or `e2`.

Mapping Program to Machine. For denoting which expressions shall be evaluated on which processors a *mapped expression* of the following form is used:

```
exp on pid
```

where `exp`, the body of the mapped expression, is an expression, `on` is a keyword and `pid` is the number of the processor on which it shall be evaluated.

Such an expression can be used for example to evaluate the two components of an addition on two different processors:

```
(f x on 0) + (g y on 1)
```

With this expression the function call `f x` will be evaluated on processor `0` and the function call `g y` will be evaluated on processor `1`. Note, that since `+` is a strict operation, both function calls will be evaluated in parallel due to the default evaluation strategy of lazy evaluation.

It is also possible to use functions in computing the processor identifier. Thereby, one can realize a mapping that is relative to the current processor (the predefined identifier `self` always contains the identifier of the current processor).

For a more detailed explanation of the concept of para-functional programming in general and especially of scheduling and mapping see [Hudak, 1989]. In this paper also some bigger examples are given that show how this concept can be used to create parallel functional programs with a special operational behaviour.

Chapter 3

ADAM & EVE

The ADAM & EVE project has been started at RISC in 1989. The goal of this project is to study the behaviour of programs in the functional programming language EVE that have been compiled to the assembler code of the abstract hybrid dataflow/controlflow machine ADAM. The project has been influenced by the work on dataflow architectures at the MIT (see [Iannucci, 1988], [Traub, 1986]). However, the ADAM & EVE project puts an emphasis on using standard hardware instead of a dedicated dataflow machine. Therefore, the first part of this project was the implementation of the abstract dataflow machine ADAM that allows to simulate dataflow behaviour on conventional hardware. The second part of the project, which is described in this thesis, is the implementation of a compiler for the functional programming language EVE that generates assembler code for the ADAM machine. This code directly reflects the structure of a dataflow-graph that can be derived from the given EVE program. Therefore, the resulting ADAM assembler program is a parallel program.

This chapter contains a condensed description of the ADAM machine and of the EVE language as it is given in [Schreiner, 1990]. However, as the EVE language has been slightly modified the language description part contains some changes to the above thesis that are explicitly emphasized.

3.1 The ADAM Abstract Dataflow Machine

The ADAM machine is an abstract hybrid dataflow/controlflow machine. The ADAM processor has a conventional two-address instruction set augmented by primitives for the creation and synchronization of parallel tasks that allow the simulation of dataflow behaviour. Special memory units on each ADAM module are responsible for the management of non-strict data structures. With this "hardware support" for non-strict data structures it becomes easier to synchronize several processes that work on such a non-strict data structure.

In this thesis, the ADAM machine will only be described as far as it is necessary to understand the meaning of an ADAM assembler program. For more information about the implementation of the ADAM machine see [Schreiner, 1990].

3.1.1 The Dataflow Principle

For explaining the basic principle of a dataflow graph let us look at the following mathematical expression:

$$(a+b)*(-y) \textbf{ where } \begin{array}{l} a=-x \\ b=x/y \end{array}$$

This expression can be translated into a *dataflow graph* by translating each operation in the expression into a node of the graph with one input arc for every operand of the operation and with one output arc. Therefore, the top-level multiplication is translated into a node with two input arcs (one for $(a+b)$ and one for $(-y)$) and with one output arc, representing the result value of the whole expression. The two input arcs have to be connected with the output arcs of the subexpressions $(a+b)$ and $(-y)$. By recursively applying this translation on all subexpressions one gets the dataflow graph of Figure 3.1.

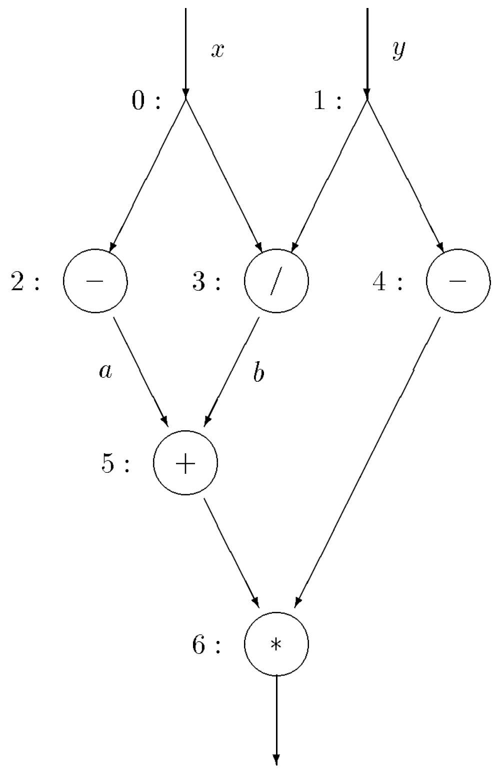


Figure 3.1: A Dataflow Graph

The whole graph has now one input arc for each free variable of the given expression. The output arc represents the result of the expression. This dataflow graph shows the data-dependencies between the various variables in the expression. Therefore, it can be used as the basis for the so called *data-driven execution model*. In this model values are represented by *tokens* that flow along the arcs of the dataflow graph and trigger the execution of the instructions in the nodes of the graph. When a token reaches a node representing a binary instruction it has to wait for the arrival of a partner token on the other input arc (*port*) of the node. As soon as both tokens are available the node "*fires*" i.e. the instruction is performed on the input tokens (thereby consuming these tokens) and a new token with the result of the instruction is created and sent away via the output arc. This is called the *firing rule* of the dataflow graph.

Looking back at Figure 3.1 we notice that as soon as a value arrives at the y input, the node number 4 in the graph may fire, sending the result to the node number 6. However, this node has to wait for the arrival of a token on the other input arc i.e. from node number 5. From the above firing rule it should be clear that this data-driven execution model bears an inherent parallelism, since the order of the evaluations in the nodes of the graph is only determined by the data-dependencies between the various objects in the graph.

0 :	COPY	2,	3.l
1 :	COPY	3.r,	4
2 :	NEG	5.l	
3 :	DIV	5.r	
4 :	NEG	6.r	
5 :	ADD	6.l	
6 :	MUL	7	

Figure 3.2: A Dataflow Program

The dataflow-graph in Figure 3.1 can be quite straightforwardly translated into a dataflow program by translating every node in the dataflow-graph into one line of the program. Each line has a number as label and contains the instruction that has to be performed and the successor instruction to which the result has to be sent (in the case of a copy instruction two successor instructions have to be included). If the successor instruction is a binary instruction, then also the port, to which the result is sent (l for "left port" and r for "right port") has to be included in the line of code. By this consideration, it is quite obvious that Figure 3.2 is the translation of the dataflow-graph in Figure 3.1. Such a program can be executed by a dataflow machine.

3.1.2 Differences between the ADAM and a Pure Dataflow Machine

One of the main disadvantages of the pure dataflow model is the fact, that many copy instructions have to be performed as it is impossible in this model to share data. Hence, whenever a token is needed by two different instructions it has to be copied. Therefore, the well known notion of a register has been introduced in the ADAM machine. This allows to replace the duplication mechanism of the pure dataflow model by sharing data that is contained in registers. However, to maintain the correct execution order an explicit synchronization instruction has to be introduced that "guards" every binary instruction as it can be seen in Figure 3.3. The first argument of this synchronization instruction (WAIT)

0 :	COPY	x_1,	x_2,	2
1 :	COPY	y_1,	y_2,	4
2 :	SPLIT	3,	6	
3 :	NEG	x_1,	x_1,	8
4 :	SPLIT	5,	6	
5 :	NEG	y_2,	y_2,	10
6 :	WAIT	e_0,	7	
7 :	DIV	x_2,	y_1,	8
8 :	WAIT	e_1,	9	
9 :	ADD	x_1,	x_2,	10
10 :	WAIT	e_2,	11	
11 :	MUL	x_1,	y_2,	12

Figure 3.3: An ADAM Assembler Program

is a so called *event*. An event has two possible states: “occurred” and “not occurred”. Initially it is set to “not occurred”. As soon as it is reached by the first thread of control (this corresponds to the arrival of a token in the dataflow model) it is set to “occurred” and this thread is blocked. When another thread of control reaches this instruction the state of the event is reset to “not occurred” and a jump to the next instruction is performed, which is usually a binary instruction. Now, it is guaranteed that both registers that are affected by the binary instruction contain values. The instruction is now performed on the two registers and the result is written to the first register, as it is done for all two-address instructions of the ADAM assembler code. Furthermore, one thread of control continues at the successor instruction, whose label is the last component of each line. Note that now the threads of control announce the availability of a value in the corresponding register, where in the pure dataflow model tokens that carry values have been sent. Summarizing one can say that in the ADAM model arcs of the dataflow graph denote registers and nodes represent two-address instructions operating on these registers.

0 :	WAIT	e_0,	1	
1 :	COPY	x_2,	x_1,	2
2 :	DIV	x_2,	y_1,	3
3 :	NEG	x_1,	x_1,	4
4 :	ADD	x_1,	x_2,	5
5 :	NEG	y_1,	y_1,	6
6 :	MUL	x_1,	y_1,	7

Figure 3.4: An Optimized ADAM Assembler Program

As one can see from Figure 3.3 the ADAM machine has a conventional two-address instruction set. The only instructions that are needed to introduce parallelism are:

- The SPLIT instruction that splits the controlflow into two parallel threads of control.

From the dataflow point of view, this instruction represents a node that takes one input token and splits it into two independent output tokens. This construct is used to create parallelism.

- The WAIT instruction that joins two parallel threads of control in the controlflow into one thread. From the dataflow point of view, this instruction represents a node that takes two input tokens and merges them into one output token. This construct is used to synchronize parallel parts of the program.

The approach taken in the ADAM machine has two important advantages in comparison to a pure dataflow machine:

- In contrast to a pure dataflow machine the ADAM machine needs not explicitly duplicate data that is referenced by more than one instruction. Instead in the ADAM machine the data is shared in common registers.
- As the synchronization is done explicitly in the ADAM machine the compiler may choose to avoid some synchronizations whenever this is safe. Thereby, the compiler can serialize the code and increase the efficiency of the whole program.

  To decide whether it is safe to serialize some parts of the code, a formal analysis of the program, called *strictness analysis*, has to be performed. The strictness analysis tells the compiler, which arguments of a function are strict and can therefore be evaluated sequentially without violating the non-strict semantics of the language.

Figure 3.4 shows an optimized version of the ADAM assembler program in Figure 3.3. As the original expression contains only arithmetical operations, which are strict in all arguments, the whole program can be evaluated sequentially. The only necessary synchronization in the whole arithmetical expression is at the begin of the program where a wait for the two free variables x and y has to be performed.

3.1.3 The Global Sructure of the ADAM Machine

In this section the global structure of the ADAM machine will be discussed as far as it necessary for understanding the operational behaviour of the ADAM machine. Especially, some aspects of the memory management have to be discussed.

The ADAM machine consists of several equally constructed worker modules that are connected by a network of channels. The machine is based on the principle of distributed memory, so information that is needed by a remote module has to be explicitly sent from the module that contains the information to the one that needs the information. There is no possibility of sharing data between several modules.

Figure 3.5 shows the structure of one module in the network. Each module consists mainly of three parts:

- **The Arbiter** is responsible for the communication between the local module and the other modules.

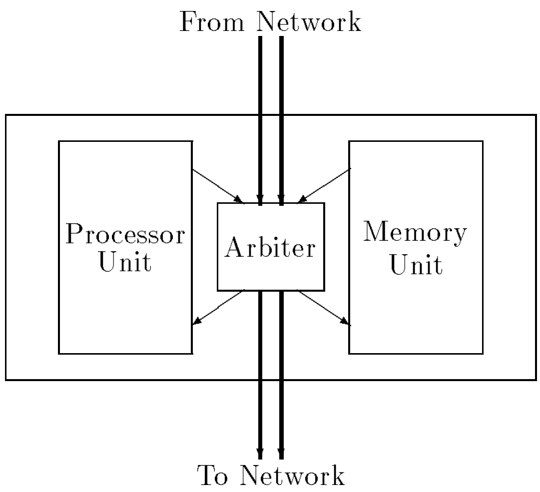


Figure 3.5: An ADAM Module

- **The Processor Unit** executes the given program and thereby performs the actual computation. The instruction set of this unit consists mainly of two-address instructions where the first argument gets the result of the computation. Therefore, the meaning of e.g.

$$\text{ADD } r_1,\ r_2,\ +1$$

is

$$r_1 \ := \ r_1 \ + \ r_2$$

and continue the execution with the next line. Apart from such arithmetical-logical instructions there are also instructions affecting the controlflow (like the well known JUMP instruction), instructions for calling functions (either first order or higher order) and instructions affecting the storage of all data structures. For a description of the meaning of all ADAM instructions see Appendix C.

- **The Memory Unit** contains a store holding tuples, i.e. data structures of variable size. For the construction of tuples, for the selection of parts of the tuple and for writing values into parts of a tuple special messages have to be sent to the memory unit. These messages and the operational behaviour of the memory unit will be discussed below.

Memory Management

The memory unit of an ADAM module deserves further examination. As the EVE language is defined to be a non-strict programming language and therefore uses non-strict data structures the ADAM machine must also be able to handle such non-strict data structures. This handling is done by the memory unit of an ADAM module.

The main problem of non-strict data structures is how to synchronize the parallel processes that want to write to or read from the data-structure. The way for solving this problem that has been chosen for the ADAM machine is the usage of so called *I-structures* as they are discussed in [Traub, 1986] and [Arvind *et al.*, 1987]. The idea of this approach is to associate with each field of the data structure a *full/empty flag*. This flag indicates, whether a value has already been written into this field or not. The status of this flag determines the action that has to be taken when one of the following messages arrives at the memory unit:

- **Allocation Request:** ADAM instruction:

  ALLOC *constant*, *label*

  A new tuple of the size *constant* is allocated and all flags of the tuple are set to empty. A reference to the tuple is returned by the memory unit and arrives at the next line in the ADAM assembler program.

- **Read Request:** ADAM instruction:

  READ *constant*, *register*, *label*

  With this instruction the value of field number *constant* of the tuple to which *register* points is read. If a read request arrives one has to distinguish two cases:

  Case flag=full: In this case the value of the requested field is returned.

  Case flag=empty: In this case the read request is put into a queue of deferred reads.

- **Write Request:** ADAM instruction:

  WRITE *constant*, *register*, *register*

  With this instruction the contents of the first *register* is written into the field number *constant* of the tuple to which the second *register* points. Again a case distinction is necessary:

  Case flag=full: This case violates the *single-assignment rule*. As this case can not occur in correct ADAM code the program is aborted in this case.

  Case flag=empty: If there is a queue of deferred reads associated to this field, then the value that will be written into the field is sent as answer to each of the read requests. Afterwards, the value is written into the specified field.

The important advantage of a non-strict data structure in comparison to a strict data structure with respect to parallelism lies in the fact that in a non-strict data structure there is a separation between the creation and the usage of the data structure. For example, after having created a non-strict data structure two references to it may be immediately sent to two different processes:

- One process (the *producer*) writes values into the data-structure,
- the other process (the *consumer*) reads values from this data-structure.

Since the data structure is non-strict, both processes can work at the same time. If the data structure were defined strict, the consumer would have to wait until the producer has computed the whole data structure. With this kind of *producer-consumer parallelism* it is possible to achieve a much higher degree of parallelism than in a strict language.

Another advantage of non-strict data structures stems from the fact, that the three ADAM instructions ALLOC, READ and WRITE are implemented as *split-phase operations*. This means that after having submitted one of the above requests the processor immediately interrupts the execution of the actual thread and starts another one. The interrupted thread is only continued at the arrival of the desired value. The advantage is now that, by this implementation, the ADAM machine tolerates extremely long memory latencies.

Another implementation detail that affects the operational behaviour of the ADAM machine is the implementation of the *garbage collector*. In the ADAM machine a *weighted reference counting algorithm* has been implemented (see also [Bevan, 1987]). The principle of this algorithm is the following: To each reference a weight and to each data structure a reference count is associated. Then, the following invariant holds: The reference count of a data structure is equal to the sum of the weights of all references to it.

To achieve this behaviour additional actions have to be taken when a request arrives at the memory unit:

- **Allocation Request:** The newly allocated tuple gets a power of two as its initial weight. It suffices to store the binary logarithm of the weight in the reference that is returned as the weight will always be a power of two.
- **Deleting References:** If a reference is not used any more it is deleted and the reference count of the referenced tuple is decreased by the weight of this reference. If thereby the reference count of the tuple becomes zero, the tuple may be reclaimed.
- **Duplicating References:** After the duplication each of the references gets half of the weight of the original reference. However, if the original reference has already weight one the reference count of the referenced tuple is set to ∞. As this reference count will never change any more the referenced tuple will never be reclaimed.

This garbage collection algorithm has the following *advantages*:

- The garbage collection is done on the fly,
- it is simple an fits very well for a parallel distributed memory system.

However, this algorithm also has some *disadvantages*:

- Cyclic structures will never be reclaimed as there is always a reference to each of the parts of the structure although the structure itself is not needed any more. This disadvantage could be overcome by implementing an additional garbage collector that reclaims such cyclic structures (e.g. a mark and scan, or a copying garbage collector) and use this garbage collector when the weighted reference garbage collector can not reclaim enough space.
- Every duplication of references involves their modification.

As we will see in the code generation step of the compiler one has to keep in mind that a tuple whose reference count drops to zero will be automatically reclaimed. And as instructions like READ and WRITE decrease the count of the structure, one has to explicitly guarantee that the reference will not be applied any more.

3.2 The EVE Programming Language

This section provides a reference manual for the functional programming language EVE. The goal in designing the language EVE was to come up with a programming language that allows an easy way of programming the ADAM machine. As it is one of the most important advantages of functional programming languages that they are highly suited to parallel computation we have chosen such a functional programming language. It would also be possible to use an imperative language that is extended by constructs for explicit parallelism. However, we prefer to use a functional programming language as it hides the details of parallelism from the programmer[1]. Therefore, EVE does not contain any constructs for explicit parallelism. Moreover, the implicit parallelism of a program will be automatically extracted by the EVE compiler (for a description how this is done see Chapter 4). The EVE languages incorporates most of the important features of todays functional programming languages but it lacks some of the constructs that can be expressed by existing features. This strategy has been chosen to make a prototype implementation of the compiler easier as the language is kept quite small.

This reference manual is based on the reference manual for EVE that has been published in [Schreiner, 1990]. However, some minor changes of the language definition have been performed when the compiler was implemented. These changes in the language definition are explicitly marked in this reference manual.

3.2.1 Characteristics of the EVE Programming Language

Here the most important characteristics of EVE are listed together with a discussion of the effects of these characteristics with respect to parallelism:

- **EVE is purely functional.**

  EVE contains no constructs that perform side-effects. Therefore, EVE if *referentially transparent*. The result value of an EVE function only depends on the values of the parameters and of the global variables that are used in the function. Variables can be viewed as names for expressions. Therefore, we do not say that a value is assigned but that it is *bound* to a variable. In all, the semantics of the programming language EVE is similar to the semantics of the usual mathematical notation.

- **EVE is non-strict.**

  In EVE a user-defined function may return a value before finishing all of its computation. Analogously, a data structure may be read before all of its components have been written into the data structure. This feature enables the user to exploit the producer-consumer parallelism, which has been described in Section 2.1.1, in a very straightforward way. For example function composition can be used to combine a producer process with a consumer process without introducing an additional synchronization point between the two processes as it would be the case in a strict programming language.

[1] For a discussion of the advantages of parallel functional programming compared with parallel imperative programming see Chapter 1.

- **EVE is strongly typed.**

  For every newly defined object an explicit type annotation has to be given. The result of this is that all types can already be checked at compile time. There is also no type inference necessary. On the other hand, this yields a monomorphic language.

- **EVE is higher-order.**

  In EVE functions are first order objects. That means that a function can be a parameter of another function, it can be the result of another function and it can be stored in a data structure.

- **EVE is safe.**

  EVE programs never produce runtime errors that force a halt of the execution. Instead of aborting the computation a special error value is returned. This is important since it might turn out that the erroneous value is not needed for the result at all. In this case the result might be valid although the computation of some part has yielded an error.

As it has already been mentioned above, EVE lacks some of the features of todays functional programming languages[2]. However, most of these features can be expressed with the existing features in EVE. The features that are not supported are:

- Pattern Matching,
- Type Inference and Polymorphism,
- Lambda Expressions and Curried Functions,
- List and Array Comprehensions,
- Delayed Evaluation,
- Abstract Data Types and Modules,
- Classes of Types.

Some remarks on the notation that is used:

- EVE program code is written in `typewriter` style e.g.

  ```
  f[x]+1 < a.
  ```

- Non-terminal symbols of the EVE grammar are written in *italic* e.g.

  Identifier.

- Comments start with a `--` symbol and range to the end of a line e.g.

  ```
  -- This is a comment.
  ```

- Note that EVE is *case sensitive*. So the following identifiers are different

[2]For a description of the most important features of functional languages see Section 2.1.

`ident`, `Ident`.

- Keywords and predefined identifiers are written in capital letters e.g.

  `IF`, `CASE`, `HEAD`, `NULL`.

3.2.2 EVE Program Structure

The overall program structure of an EVE program looks as follows:

$$\begin{array}{l} \mathit{Definition}_1 \; ; \\ \qquad\quad \ldots \\ \mathit{Definition}_n \; ; \end{array}$$

In every definition an identifier is bound to an object. Corresponding to the kind of this object one can distinguish between three different kinds of definitions:

- Type definitions,
- Constant definitions,
- Function definitions.

The details of these kinds of definitions are discussed in the next sections.

However, the prototype compiler imposes some restrictions on this general structure of an EVE program:

- *The definition of an identifier in the* EVE *program must precede its usage.* Therefore, only the identifiers that are defined in $\mathit{Definiton}_1$ – $\mathit{Definition}_i$ can be used in the body of the i-th definition. To be able to write mutually recursive programs, in which such a linear ordering of the definitions is not possible, a forward definition of types, constants and functions is possible.
- *Every identifier may be defined at most once in its level of scope.* More than one definition in the same level of scope will yield a semantic error of the compiler. But of course it is possible to use the same variable name in different blocks e.g. in different nestings of local definitions or in different functions.
- *On the top level there are no constant defintions allowed except for the last definition that must define the constant* `MAIN`. **Note:** This restriction was not contained in the previous version of the reference manual. The result value of an execution of the EVE program is the value of this constant `MAIN`. This differs to the description in [Schreiner, 1990], where a dialogue shell for calling the compiled functions is described.

Definitions

3.2.3 Type Definitions

The general form of a type definition in EVE is the following:

Ident `==` *Type*.

With this definition the identifier *Ident* is bound to the type expression *Type*. As an exception to this form a forward defintion of a type looks as follows:

Ident.

Such a definition is necessary when mutually recursive type definitions are used[3].

The types in EVE can be distinguished in atomic types and compound types. Before both kinds of types will be discussed the rule is given that decides whether two types are equal. Two types in EVE are equal, iff

- Both types are the same atomic type or
- both types are compound types with the same top level type constructor and all corresponding components of the two types are the same.

Note: Differently to the description in [Schreiner, 1990], the equality of types is based on the structure of the types rather than on the names. Therefore, this approach to the equality of types is also called *structure equivalence*. One of the advantages of this approach is that type conversions between types that have the same structure but different names become superfluous.

Atomic Types

In EVE the following atomic types are present:

- Integers (`INT`),
- Booleans (`BOOL`),
- Characters (`CHR`).

All of these atomic types are totally ordered sets.

Integers

[3] Directly recursive type definitions are also allowed. For that no forward definition is necessary.

The Data Type. The atomic type `INT` consists of all integer numbers between the two predefined constants `MININT` and `MAXINT`. The total ordering on this type is the usual ordering on integer numbers. All prefix comparison functions on integers have an `I` instead of the `*` in Table 3.6. Hence, the test if the integer 0 is smaller than the integer 1 according to the ordering on integers can be written as follows:

```
0 < 1
```

or

```
ILT[0,1].
```

The actual values of these two constants may vary between different implementations. However, the following two propositions must hold for every implementation of an EVE compiler:

- `MININT` $<$ 0 $<$ `MAXINT`,
- `INT` $==$ $\{n \mid$ `MININT` $\leq n \leq$ `MAXINT`, $n \in \mathbf{N}\}$.

If an object becomes larger than `MAXINT` or smaller than `MININT` during the computation, a special error value is returned (the same happens if a division by zero occurs). See Section 3.2.11 for a discussion of the handling of error values.

Note: In the prototype implementation of the EVE compiler the range of integer constants is even more restricted than described above. Although, the values of `MAXINT` and `MININT` that can be handled by the ADAM machine are $2^{30}-1$ and -2^{30}, respectively, every integer number in the source code must be in the range of $2^{16}-1$ and -2^{16}. This has the effect that the statement

```
n : INT == 65536
```

will yield an error by the compiler whereas the following statements are accepted by the compiler:

```
n : INT == 65535+1 ;
m : INT == MAXINT.
```

Operations on the Data Type. The operations that are defined on this atomic type of fixed-size integer values (ordered by their precedence) are shown in Table 3.1. The meaning of these operations should be obvious.

When splitting a number n into a quotient and a remainder modulo a given number m the following propositions always hold:

- $n = (n \texttt{ / } m) \texttt{ * } m \texttt{ + } (n \texttt{ \% } m)$,
- $0 \leq n \texttt{ \% } m < m$.

Operator	*Function Name*	*Description*
`-`	`NEG`	Negation (unary)
`*`	`MUL`	Multiplication
`/`	`DIV`	Division
`%`	`MOD`	Remainder
`+`	`ADD`	Addition
`-`	`SUB`	Subtraction (binary)

Table 3.1: Integer Operators

Function	*Description*
`ABS`	Absolute Value
`ODD`	Test, if number is odd

Table 3.2: Integer Functions

By default all of these binary operations are left associative. However, as in mathematical notation parentheses can be used to group subexpressions together. To distinguish the unary negation from the binary subtraction parentheses must be put around a negated expression. A valid EVE expression is then for example:

```
a-b-(-c)*(d+e).
```

If one uses the predefined functions instead of the operations one gets the following expression:

```
SUB[SUB[a,b],MUL[NEG[c],ADD[d,e]]].
```

Note: In the prototype compiler it is not possible to use any predefined function as an first order object as these functions are translated to inline code and are not present as independent functions. Therefore, the functions corresponding to the integer operations will be rarely used. However, if one needs a first order function for e.g. multiplication one has to define the function

```
Mul[x,y] : INT[INT,INT] == x*y
```

which then can be used as e.g. the parameter of a higher order function (see also Section 3.2.9 for a discussion of higher order functions).

Additionally, to the functions and operations in Table 3.1 two predefined functions that are listed in Table 3.2 can be used on integer values. The semantics of these functions is the following:

Operation	*Function Name*	*Description*
~	NOT	Negation
&	AND	Conjunction
|	OR	Disjunction

Table 3.3: Boolean Operators

- `ABS[i]` yields the absolute value of the integer `i`,
- `ODD[i]` yields `TRUE` iff `i` is odd (i.e. `i % 2 # 0`).

Booleans

The Data Type. This atomic type consists only of two distinct values that are represented in EVE by `TRUE` and `FALSE`. The ordering on this type is defined by the following rule:

`FALSE` $<$ `TRUE`.

All prefix comparison functions on booleans have a `B` instead of the `*` in Table 3.6. Hence, the test if `FALSE` is smaller than `TRUE` according to the ordering on booleans can be written as follows:

```
FALSE < TRUE
```

or

```
BLT[FALSE,TRUE].
```

Operations on the Data Type. Table 3.3 shows the available operations on boolean values and the corresponding predefined functions. Again this table is ordered by the precedence of the operations. As for the integer numbers there exists an infix and a prefix version for each of the operations on the boolean values.

One important characteristic of the functions `AND` (`&`) and `OR` (`|`) is that they are non-strict in their second arguments. Hence, if the first argument of an `AND` operation is `FALSE` then the result of the whole operation is also `FALSE` regardless of the result of the second argument (this might also be an error value). The same holds for the `OR` operation with `TRUE` instead of `FALSE`. This behaviour is also called *short circuit* discipline.

Characters

Special Character	*Description*	*ASCII Code*
`\'`	' in character constants	044
`\"`	" in string constants	034
`\\`	\ itself	092
`\b`	Backspace	008
`\f`	Form Feed	012
`\n`	Line Feed	010
`\r`	Carriage Return	013
`\t`	Horizontal Tabulator	009
`\_`	String Interrupt	—

Table 3.4: Special Characters

The Data Type. The atomic type `CHAR` contains all characters of the US-ASCII character set, which can be found in Table 3.9 in Section 3.2.13. The ordering on the characters can also be seen from this table as a character with a lower ASCII code number is smaller than one with a higher number. Therefore, when `ORD[c]` yields for a character `c` its corresponding ASCII code number then the ordering on `CHAR` can be described as follows:

Let $c_1, c_2 \in$ `CHAR`. Then $c_1 \leq c_2$ iff `ORD`$[c_1] \leq$ `ORD`$[c_2]$.

All prefix comparison functions on characters have a `C` instead of the `*` in Table 3.6. Hence, the test if the backslash character is larger than the digit 1 according to the ordering on characters can be written as follows:

```
'\\' > '1'
```

or

```
CGT['\092','1'].
```

In an EVE program a character must be enclosed in single quotation marks ('). The following characters may be used within these quotation marks:

- Any printable character from the ASCII character set (except backslash (\) and a single quotation mark (')).
- An abbreviation of the form `\l` where `l` is a character code representing a special character (e. g. `'\n'` (Newline)). Table 3.4 shows a list of all special characters that are allowed in an EVE program.
- A code sequence of the form *ddd*, where *ddd* is a three digit decimal ASCII code number from 000 to 255 (e.g. `'\036'` for `$`).

Note that the backslash character may only be used as an escape character or as the starting character of a code sequence (any other use is illegal and yields a lexical error of the compiler).

Function	*Description*
`CAP`	Capitalize a letter
`CHR`	Convert ASCII code to character
`ORD`	Convert character to ASCII code

Table 3.5: Character Functions

Operator	*Function*	*Description*
`<`	`*LT`	Less
`<=`	`*LE`	Less or Equal
`=`	`*EQ`	Equal
`#`	`*NE`	Not Equal
`<>`	`*NE`	Not Equal
`>=`	`*GE`	Greater or Equal
`>`	`*GT`	Greater

Table 3.6: Comparison Operators

Operations on the Data Type. Table 3.5 shows all predefined functions that are defined on characters. Their semantics is the following:

- `CAP[c]` yields the capitalized character of `c` if `c` is a letter, otherwise it yields `c` itself.
- `CHR[i]` yields the character that is encoded by the ASCII code of `i`.
- `ORD[c]` yields the ASCII code that corresponds to the character `c`.

Note: One change to the description of the lexical structure of EVE in [Schreiner, 1990] is the fact that now not only the symbol `#` but also the symbol `<>` can be used to denote inequality.

Compound Types

In EVE there are three *type constructors* that allow to build compound data types. These type constructors are:

- the *List Constructor* (`LIST OF`),
- the *Tuple Constructor* (`TUPLE OF`),
- the *Union Constructor* (`UNION OF`).

Lists

Function	*Description*
`::`	Construction of a new list (infix)
`CONS`	Construction of a new list (prefix)
`HEAD`	Selection of the list head
`TAIL`	Selection of the list tail
`NULL`	Test a list for emptiness

Table 3.7: Generic List Functions

The Data Type. A list is a linearly ordered sequence of values of equal type[4]. A definition of a list looks as follows:

Ident `== LIST OF` *Simple Type*

By this definiton the identifier denoted by *Ident* is bound to a list type whose elements have the type denoted by *Simple Type*. As it is not possible to define anonymous types in EVE, *Simple Type* must be the name of a previously defined type. Examples of list types are:

```
IntList  == LIST OF INT ;
Word     == LIST OF CHAR;
Sentence == LIST OF Word.
```

Operations on the Data Type. By defining a list type there is also an implicit *constructor function* of the same name available. Applying this constructor function to an arbitrary number of expressions of the correct type yields a list of this type. Some examples of objects of list type are:

```
IntList[1,2,3,4,5] ;
IntList[].
```

Note that also the empty list needs a constructor function in front of itself.

Apart from this constructor function, of which there is one for each list type, there are also some generic list functions, which operate on all kinds of lists. These functions can be found in Table 3.7. As these functions are not first-order objects, they can not be used as arguments for higher-order functions. The semantics of these functions is the following:

- The infix function `::` constructs a new list by taking its first argument as the head of the new list and the second argument as the tail of the new list. Therefore, a list that has been constructed by using a constructor function can be viewed as a short notation for a list that results from applying the `::` function on each element in a right associative way. For example, the above list of integers from 1 to 5 could also be written in the following way:

[4]Note that by this definition a list may be infinitely long.

```
1 :: (2 :: (3 :: (4 :: (5 :: IntList[]))))
```

Since `::` is right associative the parentheses can also be omitted.

- The prefix function `CONS` has the same semantics as the infix function `::`. Therefore, another possibility of denoting the above list would be to write:

```
CONS[1,CONS[2,CONS[3,CONS[4,CONS[5,IntList[]]]]]]
```

- The functions `HEAD` and `TAIL` extract the first element and the rest of a list, respectively. If the list is empty an error value is returned for both functions. As these two functions are the selector functions that correspond to the constructor function `::`, the following proposition holds for any non-empty list l:

  $l = \texttt{HEAD}[l] :: \texttt{TAIL}[l]$.

  One special feature of EVE is the following possibility to use a special syntax for denoting the application of a list selector to a list:

  List.Selector.

  This notation has the advantage that the parentheses get superfluous in a selection, which improves the readability for deeply nested selections. Compare, for example,

```
HEAD[TAIL[l]]
```

  with

```
l.TAIL.HEAD
```

  As it can be seen from this last example, the selection is left associative. But as the selection uses a postfix notation, the last expression has to be read from right to left.

- The function `NULL` tests whether a given list is empty or not. This function is very important since EVE lacks the possibility of pattern matching. Therefore, each function that operates on lists, usually must first check whether the list is empty or not[5].

[5] For examples of such functions see Section 3.2.12.

A Predefined List Type: Strings. Contrary to the three predefined types integers, booleans and characters, which are atomic types, the predefined type of strings is a compound type. *Per definitionem* a string is nothing but a list of characters. Therefore, the type of a string could be defined by the following type definition:

```
STR == LIST OF CHAR.
```

As a string is a list type, all list functions can also be applied to strings. A string constant of the form

```
"String constant"
```

is then just a shorthand for the list constant

```
STR['S','t','r','i','n','g',' ','c','o','n','s','t','a','n','t'].
```

Inside a string constant, which has to be enclosed by double quotation marks ("), the same characters are valid as in character constants with one exception:

- If a double quotation mark should be used inside the string, the escape character, which is the backslash (\), has to be put in front of it. Hence,

  ```
  "\""
  ```

  is equivalent to

  ```
  STR['"'].
  ```

Note: The splitting of strings over several lines, as it is described in [Schreiner, 1990][page 53] is not supported by the prototype compiler.

Tuples

The Data Type. Tuples in EVE represent cartesian products from set theory. Contrary to a list each tuple has a fixed size and different fields of a tuple may have different types. The general form of a tuple definition looks as follows:

$Ident$ `== TUPLE OF` $Ident_1$: $Type_1$, ..., $Ident_n$: $Type_n$

$Ident_1$, ..., $Ident_n$ are the names of the fields of the tuple and $Type_1$, ..., $Type_n$ are the corresponding types. Some examples for tuple definitions are:

```
Coordinate == TUPLE OF x : INT, y : CHAR;
Address    == TUPLE OF street : STR, number : INT.
```

The names of the fields in a tuple are all local to the tuple itself, in which they are defined. Therefore, the same name of a field can be used in different tuples. It is also possible to have the same name for a field in different levels of the tuple. But as there are no anonymous types possible in EVE, this is just a corollary from the first rule.

Operations on the Data Type. As with lists there is an implicit *constructor function* available, which has the same name as the tuple type itself. Therefore, a tuple can be constructed in the following way:

```
Coordinate[3,'A'].
```

In this object the `x`-field has the value 3 and the `y`-field has the value `'A'`.

These values can be extracted by the *selector functions* that are also implicitly available. The general form of a selector function is:

Tuple.Selector

where *Tuple* denotes an object of tuple type and *Selector* is the name of one field of that tuple. Therefore, from the above example the components can be extracted by the following expressions:

```
Coordinate[3,'A'].x ;
Coordinate[3,'A'].y ;
```

Unions

The Data Type. Unions in EVE represent disjoint unions of set theory. Hence, a union is a data structure that can be bound to values of different types, where these different types are distinguished by the tags that are associated with them. Therefore, a union can also be called a collection of types. The usage of such a type is useful when the type of an object is not known at compile time, but when there are only a fixed set of possible types. The general form of a union definition looks as follows:

$Ident$ `==` `UNION OF` $Ident_1$ `==` $Type_1$, ..., $Ident_n$ `==` $Type_n$.

$Ident_1$, ..., $Ident_n$ are the variants of the union and $Type_1$, ..., $Type_n$ are the corresponding types. Some examples for union definitions are:

```
Data    == UNION OF Num == INT, Word == STR;
Person  == UNION OF Man == Male, Woman == Female;
```

As in the tuple definition, the names of the variants are local to the definition of the union. Therefore, the same name may be used for variants in different unions.

Operations on the Data Type. Differently to lists and tuples there is no constructor function for unions. Instead, there is an implicit *conversion function* that converts an expression of a type of one variant into the union type. The general form of this conversion function is the following:

UnionType`$`*Variant*`[`*Expression*`]`

where *UnionType* is the name of the union, *Variant* is the name of a variant of this union. *Expression* must be an expression of the same type as *Variant* otherwise a type error occurs. Note that the whole expression has now the type of an union and is therefore different from the type of *Expression*. Some examples for applying a conversion function to an expression are:

```
Data$Num[317],
Data$Word["Hello"].
```

Both of these expressions have the type `Data`, which is different from `INT` and from `STR`. With union definitions also mutually recursive definitons make sense as the following example shows:

```
Subtree;
Tree    == UNION OF Node == INT, Inner == Subtree;
Subtree == TUPLE OF value : INT, left : Tree, right : Tree;
```

This example should also make clear why forward type definitions are necessary to define mutually recursive types[6].

The *selector function* for the union type is the case construct that will be discussed in Section 3.2.7.

Functions

The Data Type. In order to use higher order functions it is possible in EVE to define function types[7]. Hence, there is an own type constructor that takes $n+1$ types and yields a function type with the cartesian product of the types $2 \ldots n$ as domain and with the first type as its image. The general form of a type definition looks as follows:

Ident `==` $SimpleType_1$ `[` $SimpleType_2, \ldots SimpleType_{n+1}$ `]`

As it is described in Section 3.2.5, such an identifier that has been bound to a function type can be used in a function definition for defining the type of the function.

[6] Tuples could also be defined mutually recursive, but usually that does not make sense as contrary to a union this would yield an infinite recursion of the type.

[7] For a discussion of higher order functions see Section 3.2.9.

3.2.4 Constant Definitions

The general form of a constant definition in EVE is the following:

Ident : *SimpleType* == *Expression*.

This constructs binds the value that is the result of evaluating *Expression* to the identifier *Ident*. As EVE is strongly typed, the type of the identifier has to be given explicitly in *SimpleType*. This type must be the same as the type of *Expression*. Some examples for constant defintions are:

```
i : INT  == (2+3)*4;
s : STR  == "Concrete Mathematics";
b : BOOL == (i > 0) & ("test".HEAD = 't');
```

As the scope of an identifier starts from the point of its definition, the definition of an identifier must always precede its usage. However, it is possible to use the identifier that is just being defined inside the expression on the right hand side. Hence, the following definition is legal:

```
IntList == LIST OF INT;

ones : IntList == 1 :: ones;
```

By this definition, the identifier `ones` is bound to an infinite list, in which every element is the integer value `1`. This is an example of a directly recursive definition of a data object. However, it is also possible to give mutually recursive definitions. In this case, a forward definition has to be used to fix the type of the identifier without actually defining its value. An example of such a mutually recursive definition is:

```
IntList == LIST OF INT;

x2 : IntList;
x1 : IntList  == 1 :: x2;
x2 : IntList  == 2 :: x1.
```

This defines a list with the alternating values 1 and 2. The general form of a forward constant definition is:

Ident : *SimpleType*.

Due to the semantics of EVE as a purely functional language, each identifier may be bound at most once in one scope. If the same identifier is bound in different scopes then these are actually two different objects that happen to have the same name.

3.2.5 Function Definitions

The general form of a *function definition* in EVE is:

$Ident$ [$Ident_1$, ...,$Ident_n$] : $SimpleType$ [$SimpleType_1$, ...,$SimpleType_n$]
== $Expression$

By this definition the identifier *Ident* is bound to a function with the formal parameters $Ident_1$, ..., $Ident_n$, which have the types $SimpleType_1$, ..., $SimpleType_n$, respectively. The type of the return value is defined by *SimpleType*. *Expression* must have the type *SimpleType*. It is allowed to use global variables in *Expression*. These are variables that are different from all formal parameters and that are not bound by local definitions in *Expression*.

As it is possible to define a function type in EVE, there is a second possibility of defining a function:

$Ident$ [$Ident_1$, ...,$Ident_n$] : $SimpleType$ == $Expression$.

In this definition *SimpleType* must be a function type and the type of the return value of this function type must be the type of *Expression*.

One example for a function definition is the following definition of the famous factorial function:

```
Fact[n] : INT[INT] ==           -- Factorial
 IF n=0
   THEN 1
   ELSE n * Fact[n-1].
```

An alternative way of defining this function is to bind an identifier to this function type and then to just use the identifier in the function definition. This might be favourable if there are many functions of the same type and it is absolutely necessary if one wants to realize higher order functions[8]. Such a function definition then looks as follows:

```
IntIntFct == INT[INT];

Fact[n] : IntIntFct ==         -- Factorial
 IF n=0
   THEN 1
   ELSE n * Fact[n-1].
```

The following example shows how global variables can be used in EVE. One important feature of EVE is that it is *statically scoped*. This means that when a global variable is used the statically enclosing block of the program is searched for this variable and not the block from which the function has been called. To make this clear look at the following example:

[8] See Section 3.2.9 for a more detailed discussion of higher order functions.

```
a : INT  == 1;

f[x] : INT[INT] == a * x;

g[x] : INT[INT] ==
 { a : INT  == 2 }        -- local definition, defining a
 f[x] + 1.
```

As `a` is neither defined in the expression on the right hand side of the function definition of `f` nor a formal parameter of `f`, it is a global variable. As EVE is statically scoped the value of `a` inside `f` is 1 (the value would be 2 if EVE was dynamically scoped).

Further examples for function definitions can be found in Section 3.2.5 where the structure of an expression is discussed.

Analogously to type definitions and constant definitions it is necessary to use forward definitions if one wants to define mutually recursive functions. The general form of a function forward definition is either

Ident `[`$Ident_1$, ..., $Ident_n$`]` `:` *SimpleType* `[`$SimpleType_1$, ..., $SimpleType_n$`]`

or

Ident `[`$Ident_1$, ..., $Ident_n$`]` `:` *SimpleType* .

Such a forward definition only fixes the type of the function and the names and types of its parameters such that the function can be used in another definition. When the function is then actually defined it must have the same function head (names and types of function and formal parameters) as in the forward definition.

Expressions

After having discussed the structure of definitions in the previous section it remains to discuss the structure of an expression that appears on the right hand side of a definition.

3.2.6 Conditional Expressions

The general form of a *conditional expression* in EVE is:

`IF` $Expression_1$ `THEN` $Expression_2$ `ELSE` $Expression_3$

In this construct $Expression_1$ must have the type `BOOL` and $Expression_2$ and $Expression_3$ must have the same type. The semantics is the following: If $Expression_1$ evaluates to `TRUE` the result of the construct is the result of evaluating $Expression_2$. If $Expression_1$ evaluates to `FALSE` the result of the the construct is the result of evaluating $Expression_3$. If the evaluation of $Expression_1$ yields an error the result of the construct is `ERROR`. Due to this behaviour, the conditional construct is strict in its first argument but non-strict in the other two arguments[9].

For the implementation of this construct it is important that $Expression_1$ must be evaluated before one can decide if $Expression_2$ or $Expression_3$ yields the result. Hence, there is no speculative parallelism used in the implementation of this construct.

An example for a conditional expression is the expression defining the factorial function:

```
IF n=0 THEN 1 ELSE n * fact[n].
```

3.2.7 Case Expressions

The general form of a *case expression* in EVE is:

```
CASE Expression
     { c1 : Variant1 }          Expression1
     ...
     { cn-1 : Variantn-1 }      Expressionn-1
ELSE                            Expressionn
```

Expression must be of union type and $Variant_1$, ..., $Variant_{n-1}$ must be the name of the variants of this union type. Each of the identifiers c_1, ..., c_{n-1} scopes over its corresponding expression. The semantics of this construct is the following: *Expression* is evaluated and its value is tested by each of the branches for its actual variant (i.e. for the actual type of this value). Since the $Variants_1$, ..., $Variants_{n-1}$ must be non-overlapping at most one variant has the same type as the actual variant of this value (let this variant be the i-th variant). Now, the identifier c_i is bound to the value of *Expression*. The result of the whole construct is then the result of $Expression_i$. If no variant has the same type as the actual variant of *Expression*, then the result of $Expression_n$ is the result of the whole expression.

If the variants tested by the case construct are exhaustive i.e. if each of the variants of the union type is tested by one branch, the else branch of the construct has to be omitted.

Note: The prototype compiler does not check whether the variants are non-overlapping. If there are overlapping variants and the actual variant of *Expression* is such an overlapping variant then the first branch that has the same type as the actual variant is taken. The prototype compiler also does not check if the variants in the case construct are exhaustive. Hence, there might be an `ELSE` branch in a case construct with exhaustive variants. However, this does not change the operational behaviour of the EVE program. The prototype compiler also accepts programs with an case expression that has neither exhaustive variants nor an `ELSE` branch. In this case an error value is returned at runtime when the actual variant of the evaluated *Expression* is different from all branches in the case expression.

The following example shows how to define a union type, how to define constants of union type and how to use a case construct as a selector function on this union type by defining a function on an object of this type:

[9] For a more detailed discussion on the strictness properties of EVE see Section 2.1.1.

```
-- Type Definitions

Data  == UNION OF
           Num     == INT,
           Letter  == CHAR,
           Flag    == BOOL;

-- Constant Definitions

dInt  : Data  == Data$Num[2];
dChar : Data  == Data$Letter['a'];
dBool : Data  == Data$Flag[TRUE];

-- A case expression on that type

Invert[d] : Data[Data] ==
  CASE d
    { n : Num }    Data$Num[-n]
    { l : Letter } Data$Letter[CAP[l]]
    { f : Flag }   Data$Flag[~f] ;
```

The type definition shows how to define a collection of the types `INT`, `CHAR` and `BOOL` where the tags of these types are `Num`, `Letter` and `Flag`, respectively[10]. In the next three lines it is shown how to define constants of this union type by converting an expression of one of the three possible types into the union type. Finally, a function `Invert` is defined, that takes an object `d` of type `Data` as input and yields the "inverted" object of the same type as result. This inversion is defined differently in each of the variants of the input object:

- An integer value is inverted by changing its sign,
- a character is inverted by capitalizing it[11],
- a boolean value is inverted by applying the boolean negation on it.

After having performed this inversion, the resulting object has to be converted back to the union type `Data`. The result of the case expression (and therefore of the whole function) is therefore an object of type `Data`. Note that this case expression has no `ELSE` branch since the variants are exhaustive.

Note that the case expression is strict in *Expression* as it will be always evaluated but non-strict in $Expression_1$, ..., $Expression_n$ as only one of these expressions will be evaluated[12].

3.2.8 Block Expressions (Local Definitions)

The general form of a *block expression* (or *local definition*) in EVE is:

[10] See Section 3.2.3 for a discussion of union types.
[11] See Table 3.5 for a description of the predefined function `CAP`.
[12] For a more detailed discussion on the strictness properties of EVE see Section 2.1.1.

{
  $Definition_1$;
  ...
  $Definition_n$;
}
  Expression

In many functional languages the keyword `LET` is used instead of the symbol `{` and `IN` instead of the symbol `}`. In this construct identifiers are defined locally i.e. they scope over their own bodies, over all succeeding definitions and over the *Expression*. The value of the whole expression is the result of evaluating *Expression*. Usually, all the identifiers that are bound by $Definition_1$, ..., $Definition_n$ occur in *Expression*.

An example of such a block expression is the following:

```
{
 a : INT == 3;
 b : INT == a+2;
}
(a+b) * 7
```

As the definition of `a` precedes the definition of `b`, `a` can be used in the definition of `b`. As the values for `a` and `b` in the expression of the block expression are taken from these local definitions the whole construct evaluates to 56.

It is also possible to have nested block expressions. Since each block opens a new scope for bindings it is possible to bind an identifier that has also been bound outside (e.g. by an enclosing block expression). On the other hand, if an identifier that is used in an expression is not defined in the current block expression the enclosing block is searched for a binding of this identifier. If there is no binding in this block either, again the enclosing block of this block is searched and so on. This strategy is also called *lexical scoping*.

An example showing the features that have been discussed above is the following:

```
{ a : INT == 3 }
{
  b : INT == a;
  a : INT == 5
}
a + b
```

In this example there are two nested block expressions: The outer block binds `a` to the integer value 3, and the inner block expression binds `b` to the value of `a` in the enclosing expression (i.e. to the integer value 3) and then `a` is bound to the integer value 5. As the inner block opens a new scope, this is no redefinition. Instead, it is an ordinary binding of an identifier that happens to have the same name as an identifier in the enclosing block. Now, in the expression of the inner block the value for `a` of this inner block is taken. The result of the whole expression is therefore the integer value 8.

3.2.9 Higher Order Functions

After having discussed the basic constructs of EVE in the previous sections, now some of the more advanced features of EVE have to be described. These features are typical for functional languages and are therefore also incorporated in this very small language.

One important feature of EVE is the fact that it supports higher order functions. *Higher order functions are functions that take other functions as arguments or yield a function as a result.* This means that in EVE *functions are first order objects* i.e. they can be used in the same way as other objects. Hence, functions can also be assigned to identifiers or stored in a part of a data structure and so on.

As it has already been stated in Section 3.2.3 it is necessary to define a function type for using higher order functions. Then it is possible to define a function that has an argument of such a function type. This argument can then be applied to other objects, yielding the same result as if the function that is the value of the function type argument, was directly called with these objects.

To motivate the introduction of higher order functions into the EVE language, let us look at the following functions:

```
IntList == LIST OF INT;

IncList[l] : IntList[IntList] ==
 IF NULL[l]
   THEN IntList[]
   ELSE { hd : INT     == l.HEAD;
          tl : IntList == l.TAIL
        }
        (hd+1) :: IncList[tl];

Capitalize[str] : STR[STR] ==
 IF NULL[str]
   THEN ""
   ELSE { hd : INT     == str.HEAD;
          tl : IntList == str.TAIL
        }
        CAP[hd] :: Capitalize[tl];
```

The function `IncList` takes a list of integers and increases each of these integers by one. The function `Capitalize` takes a string, which is *per definitionem* a list of characters[13], and capitalizes this string by capitalizing each of its characters. Just from looking at the syntactic structure of these two functions it is obvious that they perform essentially the same task: They apply a certain function, increasing by one in `IncList` and capitalizing a character in `Capitalize`, to a list of the appropriate type. By using a higher order function it is possible to write a more general function that has the same structure as above but that has an additional argument of function type. This is the function that is applied to each of the elements of the list. However, as EVE does not support polymorphism, it is

[13] See Section 3.2.3 for a discussion of the data type `STR`.

not possible to write one higher order function that can then be used for both of the two examples above. Instead, one has to write one higher order function for each list type. The higher order function for `IncList` function above looks as follows:

```
IntList   == LIST OF INT;
IntIntFct == INT[INT];

IntMap[f,l] : IntList[IntIntFct,IntList] ==
 IF NULL[l]
   THEN IntList[]
   ELSE { hd : INT     == l.HEAD;
          tl : IntList == l.TAIL
        }
        f[hd] :: IntMap[f,tl];

IncList[l] : IntList[IntList] ==
 { Inc[n] : IntIntFct == n+1 }
 IntMap[Inc,l];
```

With this ability of working with functions as first order objects, one has a very powerful tool that allows to classify certain kinds of functions that have the same structure.

Note: In [Schreiner, 1990][Page 64] it is stated, that a type conversion for a function is necessary, whose type has a different name than the name of the function type in the function definition of the higher order function. However, as in this version of the language definition *structure equivalence* instead of name equivalence is used to check whether two objects have the same type, such an explicit type conversion is not necessary any more.

Note: Another difference to the language description cited above is that the prefix versions of the integer, boolean and string operators and of the comparison operators can not be used as first order objects. Hence, if one wants to use the predefined negation function `NEG` as the argument to the above `IntMap` function one has to define an own function that does the same as `NEG` and use this function as argument to the higher order function:

```
MyNeg[n] : INT[INT] == NEG[n].
```

With this function definition the following function call of the higher order function `IntMap` can be used to negate all integers in the list `l`:

```
IntMap[MyNeg,l];
```

Some exceptions from the rule that functions are first order objects have to be stated explicitly. In general, none of the predefined functions that are described in Section 3.2.3 is a first order object. This holds for the list functions `CONS`, `HEAD`, `TAIL` and `NULL` as well as for the error test function `FAILED` that will be discussed in a later section. Furthermore, all user defined data constructors (for lists, tuples and function types) are not first order.

3.2.10 Strictness Properties

For a definition of the notion of strictness see Chapter 2.1.1. There it is also explained why the strictness properties of a programming language has a strong impact on the runtime behaviour of a program.

The strictness properties in EVE can be described as follows:

- **All user defined functions** in EVE may be non-strict in all of their arguments. Therefore, the compiler can in general not evaluate the arguments of a function before the function itself. However, it would be possible to detect the strictness of some arguments of user defined functions by performing a strictness analysis on the program. The information of this analysis could then be used to evaluate the strict arguments of a function before the function itself. However, the prototype compiler does not perform any kind of strictness analysis and therefore handles all user defined functions like functions that are non-strict in all arguments.
- **All predefined functions and operators** are strict in all arguments (some exceptions are given below).
- The **application of functions** to arguments is strict in the function to be applied.
- The **boolean operators** `&` and `|` (and the corresponding prefix versions `AND` and `OR`) are strict in the first argument.
- The **list constructor** `::` (and its prefix version `CONS`) is non-strict in both arguments. The non-strictness of this function implies that all lists in EVE are non-strict. These non-strict lists are one of the central means to synchronize independent computations that may be performed in parallel by the ADAM machine.
- **All list and tuple constructor functions and all union conversion functions** are non-strict in all arguments. Therefore, not only lists but also tuples and unions are non-strict data types.
- The **conditional construct (if-then-else)** is strict in its first argument but non-strict in the other two arguments.
- The **case construct** is strict in its first argument but non-strict in all other arguments.

To make this description clearer we give here some EVE functions and analyse the behaviour of these functions with respect to termination. First of all we define the following non-terminating function

```
Forever[x]: INT[INT] == Forever[x].
```

If we additionally take the following function definitions

```
Add0[x]: INT[INT] == 1+0;   -- Not strict in x

AddX[x]: INT[INT] == 1+0*x; -- Strict in x
```

```
Branch[x]: INT[INT] ==      -- Strict in x
  IF x=0                       -- = is strict in x
    THEN 1                     -- THEN-branch not strict
    ELSE Forever[x];           -- ELSE-branch strict in x

ConsL[x]: IntList[INT] ==   -- Not strict in x
  x :: IntList[0];             -- :: not strict in first arg

ConsR[x]: IntList[INT] ==   -- Not strict in x
  1 :: IntList[x];             -- :: not strict in secnd arg
```

the termination behaviour can be expressed by the following set of equations:

`Forever[0]`	$=$	$\perp$
`Add0[Forever[0]]`	$=$	1
`AddX[Forever[0]]`	$=$	$\perp$
`Branch[0]`	$=$	1
`Branch[Forever[0]]`	$=$	$\perp$
`ConsL[Forever[0]]`	$=$	`IntList[`$\perp$`,0]`
`ConsR[Forever[0]]`	$=$	`IntList[1,`$\perp$`]`

where $\perp$ denotes a value that will never be computed. The only difference between `Add0` and `AddX` is the fact that the latter function has to evaluate its argument `x` before it can return a result[14]. This is due to the strictness of all arithmetical operations. Hence, if the evaluation of the argument for `AddX` does not terminate as in the above example, the evaluation of the function call does not terminate either. The function `Branch` simply shows the non-strictness of the if-then-else construct in its second and third argument. The last two functions show that in non-strict data structures the non-termination of the evaluation of parts of the data structure does not mean that the whole data structure is undefined. Instead, the defined parts of the data structure can be used in further computations.

3.2.11 Error Behaviour

Whenever a runtime error occurs in EVE a special value `ERROR` is returned, where `ERROR` is a generic constant i.e. it is part of every data type in EVE . This error value is handled like non-termination, which has been discussed in the previous section. The semantic counterpart of `ERROR` is therefore $\perp$.

Although, both 'non-termination' and 'error' can be denoted by the semantic object $\perp$, these two ways of yielding $\perp$ have to be distinguished in two cases:

- In all predefined binary functions and operators the 'non-terminating $\perp$' has precedence over the 'error $\perp$'. Hence, when the computation of one argument does not terminate and the computation of the other argument yields an error then the computation of the whole construct does not terminate.

[14] Of course, an optimizing compiler could easily eliminate the `0*x` part in the expression. This optimization would then change the semantics of the function `AddX`, which then would be non-strict in its argument. However, the prototype compiler does not perform any optimization.

Operator	*Error*	*Description*
`+`, `-`, `*`, `/`, `ABS`	Overflow	Result outside the range `MININT`–`MAXINT`
`/`, `%`	Zero divide	Division (resp. Remainder) by zero
`CHR`	No ASCII value	Argument outside the range 0–255
`HEAD`, `TAIL`	Empty list	Application to an empty list

Table 3.8: Runtime Errors

- The error test function `FAILED` is strict with respect to the 'non-terminating $\perp$' but non-strict with respect to the 'error $\perp$'.

Table 3.8 shows all possible runtime errors that may occur. As the error values are treated in the same way as non-termination, the strictness of a function defines if an error will be propagated or not. Hence, if a non-strict argument of a function call yields an error, then the result of the function need not be an error. But if it is a strict argument that yielded an error this error is propagated in any case.

The generic function `FAILED` is used to test whether the evaluation of an expression yields an error. Like the constant `ERROR` there is one `FAILED` function for each EVE data type.

Some examples for using this function are:

```
FAILED[ERROR]                     = TRUE
FAILED[1/0]                       = TRUE
FAILED[IntList[ERROR]]            = FALSE
FAILED[HEAD[IntList[ERROR,1]]]    = TRUE
FAILED[(1>0) | ERROR]             = FALSE
FAILED[IF ERROR THEN 0 ELSE 1]    = TRUE
FAILED[IF 1>0 THEN 0 ELSE ERROR] = FALSE
```

As it has already been discussed in the previous section, all user defined functions are non-strict in all arguments. Therefore, if we take the following function definition

```
f[x,y] :  INT[INT,INT] ==
  IF x>0
    THEN x
    ELSE y;
```

then the expression `FAILED[f[1,ERROR]]` yields the value `FALSE`.

3.2.12 Examples of EVE Programs

To show the syntax and semantics of EVE and to explain the features that have been introduced in the previous sections, some small example programs are shown in this section.

```
Drop[n,list] : IntList[INT,IntList] ==
IF ((n=0) | NULL[list])
  THEN list
  ELSE Drop[n-1,TAIL[list]] ;
```

Figure 3.6: EVE Program: Drop

```
Filter[p,list] : IntList[PredOfInt,IntList] ==
IF NULL[list]
  THEN IntList[]
  ELSE { hd : INT     == list.HEAD;
         tl : IntList == list.TAIL }
       IF p[hd]
         THEN hd :: Filter[p,tl]
         ELSE       Filter[p,tl] ;
```

Figure 3.7: EVE Program: Filter

The examples have been selected in such a way, that they show the characteristic differences to other functional languages best.

For all of these examples the following type definitions are valid:

```
IntList    == LIST OF INT;
PredOfInt  == BOOL[INT];
```

First Example: The Drop Function

Figure 3.6 shows an implementation of the function `Drop` that can be found in most functional languages. This function takes an integer value `n` and a list `l` as input. The output is the list that remains after dropping the first `n` elements of the input list `l`. If `l` has less than `n` elements then the result is an empty list.

One characteristic of EVE that can be seen from this example is that most functions that operate on lists first have to explicitly check whether the list is empty or not. This is due to the fact that there is no pattern matching in EVE.

Second Example: The Filter Function

Figure 3.7 shows a higher order function `Filter` that takes a predicate `p` and a list `list` as input and returns a list of all those elements of `list`, for which `p` holds, without changing the order of these elements.

`Filter` is a higher order function since it takes another function (`p`) as argument. The `ELSE` branch of the function shows the usage of local definitions, which are enclosed by `{` and `}`. Furthermore, it shows that the selector functions for lists (`HEAD` and `TAIL`) can be used with the slector syntax of tuples i.e. they can be put after the object itself instead of in front of it as in a function call.

References to Further Examples

In [Schreiner, 1990][Section 3.14] a bigger example of an EVE program can be found. There an algorithm for performing a symbolic differentiation is described. Among others this algorithm shows the usage of unions and of mutually recursive type definitions. However, in the definition of the function `VarEq` the type conversion is superfluous due to the new definition of the equality of types (structure equivalence instead of name equivalence).

Some further examples of EVE programs that have also been used for benchmarking can be found in Appendix D.

3.2.13 Syntax Definition

This syntax definition is taken from [Schreiner, 1990][Section 3.15] with some minor changes performed on it.

3.2.14 Notation

We define EVE 's lexical structure and syntax using a form of *Extended Backus-Naur Notation*:

Terminal Symbols (i.e. the tokens of EVE) are written in `typewriter` style.

Nonterminal Symbols (i.e. syntactic entities defined by syntax rules) appear in *italic* style.

Meta symbols (i.e. symbols that structure syntax rules) appear in simple Roman style:

- = separates a nonterminal symbol that is being defined from its definition.
- ( ) group syntactic items in order to form a larger item[15].
- [ ] enclose syntactic items that are optional, i.e. that may appear once or not at all in that place[16].
- { } enclose syntactic items that may be repeated zero or more times in that place[17].
- | separates two alternative syntactic items, one of which must appear in that place.
- . denotes the end of a syntax rule.

Each nonterminal symbol is defined by a **Syntax Rule** of the form

Nonterminal = *Definition* .

[15] Do not mix up the meta symbols ( ) with the symbols `( )` of the EVE language!
[16] Do not mix up the meta symbols [ ] with the symbols `[ ]` of the EVE language!
[17] Do not mix up the meta symbols { } with the symbols `{ }` of the EVE language!

	0	16	32	48	64	80	96	112
0	⟨*NUL*⟩	⟨*DLE*⟩	␣	0	@	P	`	p
1	⟨*SOH*⟩	⟨*DC1*⟩	!	1	A	Q	a	q
2	⟨*STX*⟩	⟨*DC2*⟩	"	2	B	R	b	r
3	⟨*ETX*⟩	⟨*DC3*⟩	#	3	C	S	c	s
4	⟨*EOT*⟩	⟨*DC4*⟩	$	4	D	T	d	t
5	⟨*ENQ*⟩	⟨*NAK*⟩	%	5	E	U	e	u
6	⟨*ACK*⟩	⟨*SYN*⟩	&	6	F	V	f	v
7	⟨*BEL*⟩	⟨*ETB*⟩	'	7	G	W	g	w
8	⟨*BS*⟩	⟨*CAN*⟩	(	8	H	X	h	x
9	⟨*HT*⟩	⟨*EM*⟩	)	9	I	Y	i	y
10	⟨*LF*⟩	⟨*SUB*⟩	*	:	J	Z	j	z
11	⟨*VT*⟩	⟨*ESC*⟩	+	;	K	[	k	{
12	⟨*FF*⟩	⟨*FS*⟩	,	<	L	\	l	|
13	⟨*CR*⟩	⟨*GS*⟩	-	=	M	]	m	}
14	⟨*SO*⟩	⟨*RS*⟩	.	>	N	^	n	~
15	⟨*SI*⟩	⟨*US*⟩	/	?	O	_	o	⟨*DEL*⟩

Table 3.9: The US-ASCII Character Set

3.2.15 Input File

Input files may consist of all printable characters of the *US-ASCII character set* (see Table 3.9). In addition, the following invisible ASCII characters *(white space)* are allowed:

- ␣ (ASCII code 032) is the *space character*.
- ⟨*TAB*⟩ (ASCII code 009) is translated to a *single space* in all places it occurs (even in text constants).
- ⟨*LF*⟩ (ASCII code 010) denotes the *end of a line*.
- ⟨*FF*⟩ (ASCII code 012) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).
- ⟨*CR*⟩ (ASCII code 013) is translated into the ⟨*LF*⟩ character in all places it occurs (also in text constants).

All other characters are *illegal* (even within text constants or comments).

3.2.16 Lexical Structure

The following syntax rules define the lexical structure, i.e. the structure of EVE tokens:

Letter	=	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z` .
Digit	=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` .
Ident	=	*Letter* { *Letter* | *Digit* }.
Integer	=	*Digit* { *Digit* }.
Number	=	*Integer* .
SpecialChar	=	*Any character of Table 3.4 except* `\'` *and* `\"` .
SingleChar	=	*Any printable character of the ASCII set including* ␣ *except* `'` *and* `\` | *SpecialChar* | `"` | `\'` .
StringChar	=	*Any printable character of the ASCII set including* ␣ *except* `"` *and* `\` | *SpecialChar* | `'` | `\"` .
Character	=	`'` SingleChar `'` .
String	=	`"` { *StringChar* } `"` .

3.2.17 Context-Free Syntax

The following syntax rules define the context-free syntax of the EVE language in four sections:

- Types,
- Definitions,
- Expressions,
- Programs.

Types

IdentList	=	*Ident* { `,` *Ident* } .
SimpleType	=	*Ident* .
TypeList	=	*SimpleType* { `,` *SimpleType* } .
Field	=	*Ident* `:` *SimpleType* .
FieldList	=	*Field* { `,` *Field* } .
Variant	=	*Ident* `==` *SimpleType* .
VariantList	=	*Variant* { `,` *Variant* } .
ListType	=	`LIST OF` *SimpleType* .
TupleType	=	`TUPLE OF` *FieldList* .
UnionType	=	`UNION OF` *VariantList* .
FunctionType	=	*SimpleType* `[` *TypeList* `]` .
Type	=	*SimpleType* | *ListType* | *TupleType* | *UnionType* | *FunctionType* .

Definition Statements

TypeDefinition	=	*Ident* [`==` *Type*].
ConstantDefinition	=	*Ident* `:` *SimpleType* [`==` *Expression*].
FunctionDefinition	=	*Ident* `[` *IdentList* `]` `:` (*FunctionType* | *SimpleType*) [`==` *Expression*].
Definition	=	[*TypeDefinition* | *ConstantDefinition* | *FunctionDefinition*].
DefinitionSequence	=	*Definition* { `;` *Definition* }.

Expressions

Expression = *ListExpr* | { *DefinitionSequence* } *Expression* | IF *Expression* THEN *Expression* ELSE *Expression* | CASE *Expression* { *Field* } *Expression* { { *Field* } *Expression* } [ELSE *Expression*].

ListExpr = *CompoundExpr* { :: *Factor* }.

CompoundExpr = *SimpleExpr* [*Relation* *SimpleExpr*].

Relation = = | # | < | <= | > | >= .

SimpleExpr = [+ | -] *Term* { *AddOp* *Term* }.

AddOp = + | - | | .

Term = *Factor* { *MulOp* *Factor* }.

MulOp = * | / | % | & .

Factor = *Number* | *Character* | *String* | *Designator* { *ArgumentList* | *Selector* } | (*Expression*) | ~ *Factor* .

Designator = *Ident* [$ *Ident*].

ArgumentList = [[*Expression* { , *Expression* }]] .

Selector = . *Ident* .

Programs

Program = *DefinitionSequence* .

3.2.18 Keywords

There are eight keywords that are reserved and must not be redefined:

```
CASE  ELSE  IF     LIST
OF    THEN  TUPLE  UNION
```

3.3 Predefined Identifiers

Predefined identifiers may be redefined by the user, although it is strongly recommended not to do so. There are three classes of predefined identifiers:

- Types,
- Constants,
- Functions.

3.3.1 Predefined Types

EVE has four predefined types that are shown in Table 3.10.

3.3.2 Predefined Constants

Table 3.11 shows the predefined constants of EVE.

The constant ERROR is a generic constant, i.e. it is element of any type and therefore compatible with every type.

Name	*Description*
INT	Integer Numbers
BOOL	Boolean Values
CHAR	Characters
STR	Strings

Table 3.10: Predefined Types

Name	*Type*	*Description*
MININT	INT	Lower bound of integer numbers
MAXINT	INT	Upper bound of integer numbers
TRUE	BOOL	Boolean constant
FALSE	BOOL	Boolean constant
ERROR	*Any*	Error value

Table 3.11: Predefined Constants

3.3.3 Predefined Functions

Functions on Atomic Types

Tables 3.12, 3.13 and 3.14 show the predefined functions working on integers, booleans and characters, respectively.

Comparison Functions

Table 3.15 describes the functions that can be used to compare objects of an atomic type (i.e. integers, booleans and characters). For each kind of comparison operator, there exist

Name	*Type*	*Description*
ADD	INT[INT,INT]	Addition
SUB	INT[INT,INT]	Subtraction
MUL	INT[INT,INT]	Multiplication
DIV	INT[INT,INT]	Division
MOD	INT[INT,INT]	Remainder
NEG	INT[INT]	Negation
ABS	INT[INT]	Absolute Value
ODD	BOOL[INT]	Is number odd?

Table 3.12: Predefined Integer Functions

Name	*Type*	*Description*
AND	BOOL[BOOL,BOOL]	Conjunction
OR	BOOL[BOOL,BOOL]	Disjunction
NOT	BOOL[BOOL]	Negation

Table 3.13: Predefined Boolean Functions

Name	*Type*	*Description*
CAP	CHAR[CHAR]	Capitalize a letter
CHR	CHAR[INT]	Character from ASCII code
ORD	INT[CHAR]	ASCII code of a character

Table 3.14: Predefined Character Functions

three functions, one for each atomic type. In order to get the actual function names, one has to replace the asterisk * in the names by `I (INT)`, `B (BOOL)`, or `C (CHAR)`.

List Functions

Table 3.16 refers to the generic type definition

AnyList `== LIST OF` *Any*.

However, all of these functions are *not* first-order objects and may not be used as arguments of higher-order functions (see Section 3.2.9).

Error Test Function

The generic error test function shown in Table 3.17 can be applied to expressions of *any* type.

Function	*Type*	*Description*
*LT	BOOL[*Atomic*,*Atomic*]	Less Than
*LE	BOOL[*Atomic*,*Atomic*]	Less or Equal
*EQ	BOOL[*Atomic*,*Atomic*]	Equal
*NE	BOOL[*Atomic*,*Atomic*]	Not Equal
*GE	BOOL[*Atomic*,*Atomic*]	Greater or Equal
*GT	BOOL[*Atomic*,*Atomic*]	Greater Than

Table 3.15: Predefined Comparison Functions

Function	*Type*	*Description*
`CONS`	*AnyList*[*Any*,*AnyList*]	Construct a list
`HEAD`	*Any*[*AnyList*]	Select the head of a list
`TAIL`	*AnyList*[*AnyList*]	Select the tail of a list
`NULL`	`BOOL`[*AnyList*]	Is a list empty?

Table 3.16: Predefined List Functions

Function	*Type*	*Description*
`FAILED`	`BOOL`[*Any*]	Test for `ERROR`

Table 3.17: Error Test Function

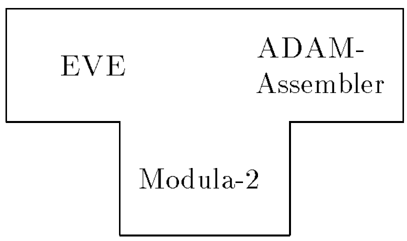
However, `FAILED` is *not* a first order object and may not be used as an argument of a higher-order function (see Section 3.2.9).

Chapter 4

The EVE Compiler

In this chapter the details of the prototype EVE compiler are discussed. The source language of the compilation process (EVE) and the abstract target architecture (ADAM) have been described in the previous chapters.

The prototype compiler can be characterized by the following T-diagram:



Source Language: The source language of the compilation process is the functional programming language EVE that has been described in detail in Section 3.2. As EVE is a functional programming language, many techniques that have been developed for a sequential implementation of such languages can also be used here. One such technique is the λ-lifting process that will be discussed in Section 4.3 of this chapter. However, as EVE is a quite small language some more complex topics of the implementation of functional languages are not covered by the compiler. For example, there is no handling of pattern matching required. For a study of the translation of these additional language constructs the interested reader may refer to [Peyton Jones, 1987].

Target Language: The target language of the compilation process is the assembler code of the ADAM abstract dataflow machine. The architecture of this machine has been discussed in Chapter 3.1. However, for the compilation process only the assembler language of the ADAM machine is important. For a short description of the assembler code see Appendix C.

Implementation Language: As implementation language for the prototype EVE compiler Modula-2 has been chosen. With this language it is quite easy to separate the compiler in its different components. Furthermore, this language supports the implementation of data encapsulations where an opaque[1] type is defined together with all procedures that may operate on this type. As the realization of the type is not visible from outside no direct manipulations of this type can be performed. To be more exact concerning the language, the JPI version of Modula-2 with the TopSpeed[2] Modula Compiler of Jensen & Partners, International is used for the implementation. This JPI Modula contains some enhancements to standard Modula-2. The whole compiler has been developed under DOS. For a description of Modula-2 in general see [Blaschek *et al.*, 1986] and [Wirth, 1988]. For a description of JPI Modula see [Jensen, 1988] and [King, 1988].

4.1 The Structure of the Compilation

The tasks that must be performed by the EVE compiler are the following:

- Lexical Analysis,
- Syntax Analysis,
- Generation of an abstract syntax tree,
- Transformations on that abstract syntax tree,
- Code Generation.

Note that in this prototype compiler no optimizations are performed. For a discussion of possible sources of optimizations see Chapter 5.

For performing all these tasks the whole compiler is split into three passes. The work that is done in the different passes is the following:

- The **first pass** performs the *lexical analysis*, the *syntax analysis* and creates an abstract syntax tree.
- The **second pass** then performs some *transformations on the abstract syntax tree*. The most important transformation is the λ-lifting process. After having performed these transformations, all functions are on top level i.e. there are no nested functions any more. This makes the code generation significantly easier.
- The **third pass** *generates* ADAM *assembler code* from the abstract syntax tree and other intermediate data structures of the compiler. This pass has to create a program file, containing for each EVE function the corresponding ADAM code, and an expression file, containing the code for evaluating the result of the whole EVE program. Furthermore, a data description file has to be produced that describes the structure of the result type of the program. This file can be already generated after the first pass. For a more detailed description of the structure of these files see Appendix C.

Figure 4.1 shows this structure of the whole compilation process.

---

[1] An opaque type is a type whose name is declared in the definition module but whose realization is declared in the implementation module, thereby hiding the information about the internal structure of the type.

[2] TopSpeed is a trademark of Jensen & Partners International.

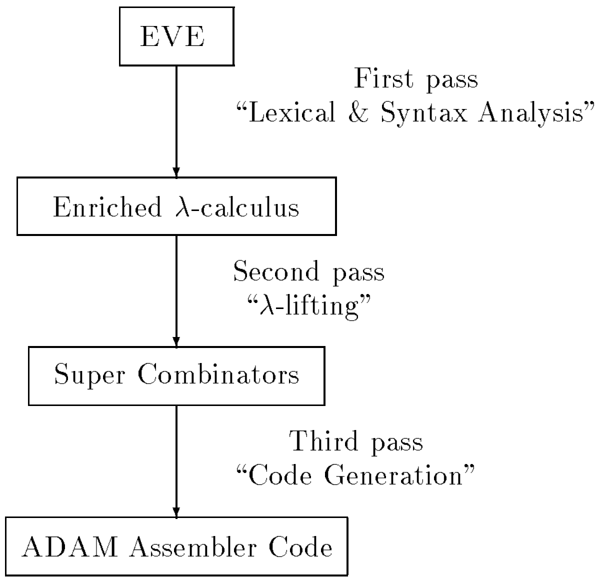


Figure 4.1: Structure of the Compilation

In translating an EVE program into ADAM assembler code, the compiler needs a lot of intermediate data structures. As these intermediate data structures are realized as abstract data types, they are implemented via *data encapsulations*. Such a data encapsulation declares a type that uniquely corresponds to the abstract data type together with some procedures that perform operations on this data structure. However, the internal structure of this type is not visible from the outside and therefore all operations on objects of this data type have to be performed via these procedures. This principle allows to change the implementation of an abstract data type without changing the code that uses this type.

The modules of which the compiler consists can be split into several classes:

- One class of modules consists of the top level module for the whole compiler and the *top level modules* for each of the tasks that have been described at the beginning of this chapter. The modules of this class can be found in Table 4.1.
- Another class of modules is formed by all *data encapsulations* for the intermediate data structures of the compiler. Table 4.2 shows the modules that fall into this class.
- Finally, there is one class with some *special modules*. The modules that handle the I/O belong to this class. They can be found in Table 4.3.

For a more detailed description of the data encapsulations see Appendix E where also the documented definition modules of all modules of the EVE compiler can be found. However, the source code of the compiler (about 20000 lines) is not included in this thesis.

Module Name	*Description of the Module*
`EveCompile`	Top level module of the whole compiler
`EveLexAn`	Module for lexical analysis
`EveSyntAn`	Module for syntax analysis
`EveLambdaLift`	Module for λ-lifting
`EveCodeGen`	Module for code generation

Table 4.1: Top Level Modules

Module Name	*Abstract Data Type*
`EveAST`	Abstract syntax tree
`EveCode`	Code area
`EveEnvironment`	Environment for code generation
`EveEvent`	Events for code generation
`EveLabel`	Labels for code generation
`EveList`	Lists of numbers
`EveListList`	Lists of lists of numbers
`EveRegister`	Registers for code generation
`EveStringLib`	Strings
`EveSym`	Symbol table and symbol list
`EveType`	EVE data types

Table 4.2: Data Encapsulations

Module Name	*Description*
`EveCoIO`	Handling of result files
`EveError`	Error handling
`EveGetChar`	Elementary input
`EveWrite`	Output of enumerations and sets

Table 4.3: Special Modules

4.2 First Pass

The first pass of the EVE compiler does not vary significantly from usual compilers for imperative languages. This is not very surprising since the analyses in this first pass perform just a syntax check and a syntactic transformation of the program. In the first pass the following things have to be done:

- A **lexical analysis** has to be performed, guaranteeing that all terminal symbols and members of terminal classes of the EVE grammar are correctly built. Among others this includes checks for the size of constants in the program, for the correct structure of identifiers and so on. The top level module for the lexical analysis is `EveLexAn`.

- A **syntax analysis** has to be performed to guarantee that the syntactic structure of the given EVE program is correct. For this analysis the principle of the *recursive descent* has been used. However, before being able to use this method the given EVE grammar of Section 3.2.13 has to be transformed into an equivalent grammar that does not include same terminal starts of different alternatives in a rule and that does not contain left recursion. This transformed grammar can be found in Appendix A. The top level module for the syntax analysis is `EveSyntAn`.

- An **abstract syntax tree** has to be built. This abstract syntax tree directly reflects the structure of the program but the tree is easier to handle by other passes of the compiler than the original source code. The handling of this abstract syntax tree is performed by an own data encapsulation in `EveAST`. This tree will be used through all other passes of the compilation.

- A **symbol table** and a **symbol list** have to be created. The difference between these two data structures is the following: The symbol table contains all object of the entire program. After an object has been added to the symbol table this entry will not be deleted until the whole symbol table is not needed any more. The symbol table is needed for the first pass and the second pass, but it can be deleted before the third pass. On the other hand, the symbol list contains only those object that are at certain moment in the scope of the program[3]. Hence, the symbol list has to be handled in a stack wise fashion. Contrary to the symbol table, the symbol list is only needed in the first pass. Both data structures are put into one data encapsulation, namely in `EveSym`.

- The **types** of the EVE program are handled in an own data encapsulation `EveType`. There the representations of the types that are defined in the EVE program are built, the equality of types is checked and so on.

In the following sections each of these steps will be discussed in more detail.

4.2.1 Lexical Analysis

[3] Actually, the symbol list contains only pointers into the symbol table, where the full information of all objects is stored.

Symbol	*Description*	*Attributes*	
`char`	Character	`charVal`	ASCII value of the character
`ident`	Identifier	`spix`	Index of identifier in the namelist
`number`	Integer number	`value`	Value of last number
`string`	String	`saddr`	Index in the string storage
		`slen`	Length of the string

Table 4.4: Terminal Classes and their Attributes

The goal of this analysis is to combine single characters of the source program to symbols. These symbols may be terminal symbols of the EVE grammar or they may be members of *terminal classes*. Such a terminal class is a set of strings that are all built according to the generation rule that characterizes that class.

Terminal symbols and terminal classes in EVE are represented by the enumeration `Symbol`. Table 4.5 shows the representations of all terminal symbols in EVE. Contrary to these terminal symbols, members of terminal classes have attributes. These attributes vary between different terminal classes. Table 4.4 shows the representations of terminal classes and the attributes they need.

In order to avoid creating a procedure that has a variable of type `Symbol` and all attributes as arguments, although not all attributes are needed normally, there are global variables in the module `EveLexAn` that always refer to the last symbol in the input file. These global variables are only set in the procedure `NewSym`, which yields the next symbol of the source program and puts it into the global variable `sy`. For a description of these global variables see the description of `EveLexAn` in Appendix E.

Renaming of Identifiers

In EVE it is possible to have two different identifiers with the same name in different blocks. However, during the lexical analysis the variable names are made unique. For that purpose a procedure `Rename` is used that substitutes a name that has already appeared in a different block by a unique name. Such a renaming has to be initiated by the syntax analyzer when in a definition of an identifier it is detected that this identifier already exists in the symbol list.

At some points the compiler has to create new identifiers that did not appear in the source code. For performing this task there is an own procedure `GetNewIdent` in the data encapsulation of the symbol table (`EveSym`) that yields a new and unique identifier.

Internally, the lexical analyzer is split into four different local modules:

- The module `EveNames` that handles the identifiers in EVE as well as all keywords. For that a special `namelist` has to be used that contains all the names of the identifiers in the EVE program. The keywords are put at the beginning of that list. For handling the renaming of identifiers correctly an additional list `renamings` has to be introduced

Symbol	*Description*
`noSy`	Dummy symbol
`eofSy`	End of file reached
`caseSy`	Keyword CASE
`elseSy`	Keyword ELSE
`ifSy`	Keyword IF
`listSy`	Keyword LIST
`ofSy`	Keyword OF
`thenSy`	Keyword THEN
`tupleSy`	Keyword TUPLE
`unionSy`	Keyword UNION
`divSy`	Integer division (`/`)
`minusSy`	Integer subtraction (`-`)
`modSy`	Integer remainder (`%`)
`plusSy`	Integer addition (`+`)
`timesSy`	Integer multiplication (`*`)
`andSy`	Boolean conjunction (`&`)
`notSy`	Boolean negation (`~`)
`orSy`	Boolean disjunction (`|`)
`eqSy`	Comparison for equal (`=`)
`geSy`	Comparison for greater equal (`>=`)
`gtSy`	Comparison for greater (`>`)
`leSy`	Comparison for less equal (`<=`)
`ltSy`	Comparison for less (`<`)
`neSy`	Comparison for not equal (`#` or `<>`)
`lbraceSy`	Left brace (`{`)
`lbracketSy`	Left bracket (`[`)
`lpar`	Left parenthesis (`(`)
`rbraceSy`	Right brace (`}`)
`rbracket`	Right bracket (`]`)
`rpar`	Right parenthesis (`)`)
`colonSy`	Separation symbol (`:`)
`commaSy`	Grouping symbol (`,`)
`conversionSy`	Conversion symbol for unions (`$`)
`defineSy`	Definition symbol for binding (`==`)
`doublecolonSy`	Infix list constructor (`::`)
`selectionSy`	Field selection from a tuple (`.`)
`semicolonSy`	Definition separation symbol (`;`)

Table 4.5: Representation of Terminal Symbols

that connects the non-unique names of the identifiers in the EVE source program with the internal unique names. So, this local module can be regarded as a data encapsulation of the list of all names. With the procedures of this module it is now possible to use numbers instead of strings as the names of identifiers. The global variable `spix` that contains the spelling index of the last identifier in the program is only set in this local module.

- The module `EveSkipComment` handles the comments in EVE. One comment in EVE always starts with a `--` symbol and extends to the end of the line but not further.
- The module `EveNumbers` handles all integer constants in the EVE program. The maximal value of such a constant that is accepted by this module is the exported constant `Maxint`. The global variable `value` that contains the value of the last integer constant in the program is only set in this module.
- The module `EveStrings` handles all string constants in the EVE program. For that a local string storage is necessary. So, this local module can also be regarded as a data encapsulation of this string storage. In a similar manner as it is done with the identifiers, every string constant is uniquely determined by some global variables. These global variables are: `saddr` containing the address of the first character of the string in the string storage and `slen` containing the length of the string. Both global variables are only set in this local module.

For details of the lexical analysis see the comments in the definition module of `EveLexAn` in Appendix E.

4.2.2 Syntax Analysis

The whole syntax analyzer is based on the principle of the *recursive descent*. For a description of this principle see [Aho *et al.*, 1986]. However, a grammar that shall be implemented via a recursive descent parser must not contain any

- left-recursion or
- alternatives with same terminal starting symbols.

As the EVE grammar of Section 3.2.13 has alternatives with same terminal starting symbols (e.g. in the definition of the non-terminal symbol *Type*) it has to be transformed into an equivalent grammar that does not contain alternatives with the same terminal starting symbol any more. This transformation is performed by applying a *left-factorization* on the critical definition. In this left-factorization the common starting sequences of the alternatives are put in front of a new non-terminal symbol. This new symbol defines the uncaptured rest of the original non-terminal symbol and does not have same terminal starting symbols any more. For a more detailed explanation of the general process of left-factorization see again [Aho *et al.*, 1986][Page 178f]. The new non-terminal symbols that have to be introduced into the EVE grammar are: *SimpleOrFctTyp* and *TCFDefinition*. The transformed grammar, which can be implemented by using a recursive descent parser, can be found in Appendix A.

Error Recovery

One important point in the syntax analysis is the way how an error recovery is performed. Many ways how to realize an error recovery have been suggested in the literature. As the EVE compiler is just a prototype compiler, we have chosen a quite simple strategy that is powerful enough to allow an error recovery after a few symbols in most cases.

The error recovery that is used here is based on the principle of the *specific stop-symbols*[4]. The idea of this principle is to pass to each procedure that parses a non-terminal symbol, a set of so called stop-symbols. When a syntax error occurs in this procedure all symbols in the source code up to the first stop-symbol are skipped. As this set is different for every procedure this is a very flexible way to handle error recovery. Usually when a procedure is called it gets the set of stop-symbols of the caller and some additional symbols that can be added to this set as we know the context more exactly now. However, one disadvantage of this method is that as each procedure that parses a non-terminal symbol needs an additional parameter, the compilation is slowed down by the passing of the additional parameter a little even when no error occurs.

Although this principle usually yields a quite good error recovery, problems may arise when there are many syntactic and semantic errors mixed in an EVE program. In such a case some internal data structures of the compiler might be not defined (as the corresponding EVE construct yielded an error). Therefore, it is possible that the compiler terminates with a special message in this case.

Output of the Syntax Analysis: Abstract Syntax Tree

One output of the syntax analysis is an abstract syntax tree that represents the EVE source program. The tree structure itself is hidden in the data encapsulation `EveAST`. Hence, the structure of the tree will be discussed in Appendix E where this data encapsulation is discussed. Furthermore, in this appendix some procedures of the data encapsulation and their usage in the syntax analyzer will be discussed.

For each kind of node in the abstract syntax tree there is a constructor function that takes the components of the subtree that has to be built as arguments and creates the appropriate subtree. The result of each of these abstract syntax tree constructor functions is an abstract syntax tree (the type is therefore `ASType`) and it is returned as last argument of the procedure. The names of these constructor functions all have the form `MakeAST*`.

Every non-terminal symbol of the EVE grammar has a synthesized attribute (propagating upward) containing the abstract syntax tree of the EVE code that corresponds to that non-terminal symbol. The synthesized attribute of *Program* is then the abstract syntax tree of the whole EVE program. This attribute is also the only argument of the syntax analyzer (`SyntAn`).

Output of the Syntax Analysis: Symbol Table

For storing the objects of an EVE program together with its attributes there are two data structures required:

[4] For a discussion of the method of error recovery see [Wirth, 1984].

- A symbol table, which contains at the end of the syntax analysis all objects of the EVE program such that later passes can access these objects.
- A symbol list, which only contains those objects that are in the current scope of the program. Actually, the symbol list contains only pointers to the symbol table, where the actual information about the objects is stored. This data structure is necessary to handle the block oriented semantics of EVE correctly. At the end of the syntax analysis the symbol list is empty and not needed any more.

Both of these data structures are encapsulated by the module `EveSym`. The most important procedures that operate on these data structures can be divided into several classes:

- Procedures for adding EVE objects to the symbol table and to the symbol list.
- Procedures searching for specific entries in the symbol list.
- Procedures for explicitly setting certain fields of entries in the symbol table.
- Procedures for handling forward definitions.
- Procedures for extracting information about functions from the symbol table.
- Procedures for handling blocks in EVE .
- Procedures performing explicit operations on the constant `MAIN`.
- Procedures that are needed for λ-lifting.

Output of Syntax Analysis: Types

As EVE is strongly typed, each new identifier has to get an explicit type. However, it is also possible in EVE to apply type constructors to already existing types, thereby getting new types[5]. For creating such new types and for accessing elements of these types the data encapsulation `EveType` has been introduced.

Whenever a new compound data type is defined, this type is stored in an object of type `EType`. This new type is then stored in the type field of the object to which the new type is bound. Contrary to the symbol table there is no own data structure that contains all the types of the EVE program. Moreover, the types are only accessed as fields in the symbol table. As the types of the EVE objects are only needed in the first pass and in the second pass, they can be discarded before the code generation.

4.3 Second Pass

The second pass of the compiler contains several transformations that are performed on the abstract syntax tree that is the result of the first pass. The most important of these transformations is the λ-*lifting*. Because of its importance we first discuss the general idea of λ-lifting before we describe the λ-lifting algorithm that has been used in the EVE compiler.

[5] See Section 3.2 for a description of the available type constructors in EVE .

4.3.1 The Principle of λ-lifting

The motivation for performing a λ-lifting is the observation that functions without global variables are very easy to handle. The reason for that is that without global variables no environment that would contain the values of the global variables for a function is needed. The approach of introducing such environments leads for example to the SECD machine that is described in [Henderson, 1980]. In traditional implementations of imperative languages one also has such an environment, which is represented by the stack with the activation records of all active functions. However, the usage of such an environment slows down the computation very much. Therefore, it is worthwhile to transform the given program into a program that has no functions containing global variables any more. This process is then called λ-lifting. After this transformation all function definitions can be put on top level. These functions are then called *super combinators*. The notions of super combinators and λ-lifting have both been introduced by R. J. M. Hughes in [Hughes, 1984]. The following definition of a super combinator is taken from [Peyton Jones, 1987][Page 223]:

Definition 2 (Super Combinator) *A* super combinator*, S, of arity n is a lambda expression of the form*

$$\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$$

where E is not a lambda abstraction[6] *such that*

1. *S has no free variables,*
2. *any lambda abstraction in E is a supercombinator,*
3. *$n \geq 0$; that is, there need be no lambdas at all.*

Another way of using combinators to implement functional programs is to use a fixed set of combinators. This approach has been chosen for example by D. Turner in his implementation of SASL ([Turner, 1979]) and of Miranda[7]. In [Stoye, 1985] and [Clarke *et al.*, 1980] it is shown how this approach can be used to develop a processor that uses a fixed set of combinators as its instruction set. This possibility of building a processor that implements these combinators in hardware is one of the main advantages of this approach. As it suffices to have the S and the K combinators[8] from combinator theory such machines are usually called SK reduction machines.

Contrary to this approach of SK combinators we use super combinators in the EVE compiler. Note, that the S and K combinators are just special cases of super combinators. The most important features of super combinators, especially in comparison to SK combinators, can be summarized as follows:

---

[6] This just ensures that all the 'leading lambdas' are accounted by $x_1 \ldots x_n$.

[7] Miranda is a trademark of Research Software Ltd.

[8] The definitions for these combinators are:

- S f g x = f x (g x),
- K c x = c

- When a super combinator is applied to some expression all formal parameters of the super combinator can be instantiated by these expressions simultaneously instead of instantiating them one by one.
- As for each program a special set of super combinators is produced, there are fewer combinator applications necessary than with the SK approach. Hence, one could say that super combinators have a 'coarser grain size' than SK combinators. This offers some further advantages like less complex compilers that yield smaller programs.
- The λ-lifting step, which transforms a functional program into a set of super combinators, can be performed in such a way that the resulting collection of super combinators is *fully lazy*. This means that when the transformed program is executed, no expression will be evaluated twice. To reach such a behaviour, the λ-lifting has to lift so called *maximal free expressions* (*mfe*'s) instead of single global variables of functions. A maximal free expression of an expression E is a free expression that is not a proper subexpression of another free expression of E. In the above mentioned original work of Hughes ([Hughes, 1984]) such a fully lazy λ-lifting was developed. For another discussion of fully lazy λ-lifting see [Peyton Jones, 1987][Chapter 15]. Note that such a fully lazy behaviour of the resulting program is also possible when the given program is compiled to SK combinators.

Apart from the λ-lifting algorithms that are presented in [Hughes, 1984] and in [Johnsson, 1987] there are other algorithms known in the literature that put an emphasis on different aspects of the whole process. For example in [Takeichi, 1988] an algorithm for the so called 'lambda-hoisting' is presented. The key idea behind this algorithm is to transform programs into ones with local recursion. Hence, locally defined functions are not lifted on top level as they are by the other λ-lifting algorithms. Furthermore, the lambda-hoisting algorithm is fully lazy. The main advantage of this algorithm is that deeply nested functions get fewer parameters than with the other λ-lifting algorithms. This algorithm has been used to compile code for the 'fully lazy functional machine' (FLFM), which is described in [Takeichi, 1986]. A modified version of this machine has been used for the implementation of the functional programming language RUFL (see [Wentworth, 1989b]).

The main new idea that is presented in [Peyton Jones and Lester, 1991] is to separate the process of a fully lazy λ-lifting into two steps:

1. A full laziness transformation and
2. an ordinary λ-lifting step and

The *full laziness transformation* itself can again be decomposed into two steps:

1. In the first step each *maximal free expression* e is replaced by the local definition $\{ \ v == e \ \} \ v$.
2. Afterwards, all local definitions are lifted to the top level.

This approach has several advantages compared to the other approaches discussed so far:

- Compared to the algorithms performing a fully lazy λ-lifting the single steps in this approach are quite simple. This reflects the general software technological idea of splitting big problems into smaller ones.
- For the λ-lifting steps different algorithms can be taken without having to change other parts of this algorithm. Thereby, one can make experiences with already existing λ-lifting algorithms that do not keep full laziness.
- The full laziness transformation can even be used in compilers that do not perform any λ-lifting at all. Hence, it might be useful to study the advantages of performing such a transformation in its own right.

This λ-lifting algorithm is used in the implementation of the functional programming language Haskell at the University of Glasgow. Therefore, the description of the algorithm itself is given in Haskell, which forms a nice example of how to structure programs in a modern functional programming language.

4.3.2 The λ-lifting Algorithm in the EVE Compiler

The λ-lifting algorithm that is used in the EVE compiler is similar to the algorithm described in [Johnsson, 1987]. As this is no fully lazy λ-lifting algorithm no maximal free expressions have to be detected. Instead, it essentially suffices to transform every global variable of a function into an additional parameter. The resulting functions can then be put on the top level. There are two steps in the algorithm that can be described as follows:

- For each free variable in a local function definition, add an argument to the function.
- Apply the same free variables to each use of the function, substituting $fx_1 \ldots x_n$ for f, where $x_1 \ldots x_n$ are the free variables in the definition of f. The other actual parameters stay at the end of the function call.

To make this strategy clearer let us look at the following example:

```
f[x] : INT[INT] ==
  { y    : INT      == 2*x ;
    g[x] : INT[INT]  == x+y }
  g[x*x];
```

In this example the variable `y` is a global variable for the function `g`. Therefore, the definition of `g` is extended by one argument. This argument stands for the global variable `y`. In the call of `g` an additional argument has to be given. It should be obvious that this additional argument must be the global variable in the above example. After these two steps one gets the following lifted version:

```
g[x,y] : INT[INT,INT] == x+y;

f[x] : INT[INT] ==
  { y : INT == 2*x }
  g[x*x,y];
```

Now, the two functions f and g are super combinators as they don't have any free variables any more.

From this example one could get the impression that it suffices to determine the set of free variables for each function and then add these free variables to every call of the function and add a formal parameter for each free variable in the function definition. However, when other functions are called in the current function then it might be the case that for these functions variables have to be lifted. These lifted variables must be added to the set of lifted variables of the current function, as long as these variables are not bound inside the current function. Hence, the whole λ-lifting process has to be performed recursively. Again, to make this clearer we given an example:

```
f[x] :  INT[INT] ==
  { a : INT == 1 ;
    b : INT == 2 ;
    g[y] : INT[INT] == IF y>a THEN 1 ELSE a+h[y+1] ;
    h[z] : INT[INT] == IF z>b THEN 1 ELSE b*g[z+1] }
  g[0]+h[1];
```

If we compute the set of free variables for both local functions we get:

$$\begin{aligned} \mathit{freeVars}(\mathtt{g}) &= \{\mathtt{a}\}, \\ \mathit{freeVars}(\mathtt{h}) &= \{\mathtt{b}\}. \end{aligned}$$

But if we then lift only these free variables, the resulting function definitions again have free variables:

```
f[x] : INT[INT] ==
  { a : INT == 1 ;
    b : INT == 2 ;
    g[y,a0] : INT[INT] == IF y>a0 THEN 1 ELSE a0+h[y+1,b] ;
    h[z,b0] : INT[INT] == IF z>b0 THEN 1 ELSE b0*g[z+1,a] }
  g[0,a]+h[1,b];
```

Since the free variable b was lifted out of the function h it suddenly appears in the call of h in the body of g. Hence, b is now a free variable of g, which was not the case in the original program. The same thing happens with the variable a in the function h. Therefore, the above step has to be repeated with the new sets of free variables. This then indeed yields functions without any free variables that can be put on top level:

```
g[y,a0,b0] : INT[INT] == IF y>a0 THEN 1 ELSE a0+h[y+1,b0] ;
h[z,b0,a0] : INT[INT] == IF z>b0 THEN 1 ELSE b0*g[z+1,a0] ;

f[x] :  INT[INT] ==
  { a : INT == 1 ;
    b : INT == 2 }
  g[0,a,b]+h[1,b,a];
```

This example shows, that the set of variables that have to be lifted out of a function is not always equal to the set of free variables of that functions. Moreover, all variables that have to be lifted from functions that are called from inside these functions have to be lifted, too. This yields the following recursive definition of the set of variables that have to be lifted out of a function `f`:

$$\mathit{liftedVars}(f) \;=\; \mathit{freeVars}(f) \;\cup \bigcup_{g\ \mathit{occurIn}\ f} \mathit{liftedVars}(g),$$

where

$$g\ \mathit{occurIn}\ f \text{ iff } f \text{ contains a function call of } g.$$

If one lifts for each function f the set of variables denoted by $\mathit{liftedVars}(f)$ instead of $\mathit{freeVars}(f)$ then it is not necessary to repeat the lifting several times until there is no function with free variables any more like it was the case in the above example. Therefore, it is far more efficient to first compute $\mathit{liftedVars}(f)$ for each function f and afterwards lift this set in one step.

To make the computation of $\mathit{liftedVars}(f)$ easier we can give a different but equivalent definition of this set:

$$\mathit{liftedVars}(f) \;=\; \bigcup_{g\ \mathit{occurIn}^*\ f} \mathit{freeVars}(g),$$

where

$\mathit{occurIn}^*$ is the reflexive, transitive closure of the $\mathit{occurIn}$ relation.

The Steps of the λ-lifting Algorithm in the EVE Compiler

After having discussed the general idea of λ-lifting in the previous section, we can now have a closer look on the internal structure of the λ-lifting algorithm in the EVE compiler. The λ-lifting is done in the module `EveLambdaLift`. However, there are also functions from `EveAST` and `EveSym` used in this algorithm. The whole λ-lifting process is separated into five steps:

1. First of all, the whole *symbol table is searched for all local functions*. For each of these functions, a list of all free variables, a list of those functions that appear in this local function and a list of variables that are defined in this local function has to be created. As all this information is derived from the symbol table, this step is realized by calling the procedure `GetLists` of the data encapsulation of the symbol table (`EveSym`).

2. In the next step the *transitive closure* of the "occurIn" relation, which has been described in the previous section, has to be computed. The result of this step is for each local function a list of those variables that have to be lifted out of the function. The computation of the transitive closure is performed in a local module of the module `EveLambdaLift`.

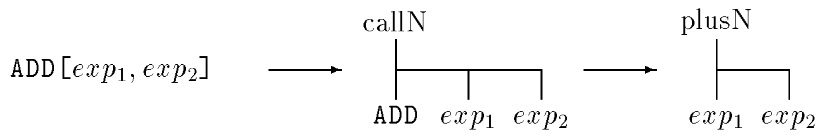


Figure 4.2: Translation of a Call of a Predefined Function into a Special Node of the Tree

3. The third step of the λ-lifting algorithm again operates on the symbol table. In this step for each local function the variables that have to be lifted are *added as parameters to the function definition*. Furthermore, all these local functions are put into a list that is then used in step five to bring all local functions on top level. After this step no function contains free variables any more. The procedure that does all this is `LiftFcts` in the data encapsulation of the symbol table (`EveSym`).

4. In this step the abstract syntax tree is traversed. For that, the procedure `Traverse` from the data encapsulation of the abstract syntax tree (`EveAST`) is used. When traversing the tree there are two things that have to be done:

   - The *local function definitions* have to be *cut off* the tree. This can be done here as they have been put into a list in step three.
   - In every *function call* the *lifted arguments* of that function have to be *added*. Note that for the function definitions these arguments have already been added in step three.

5. As the last step in the λ-lifting algorithm the *local functions* that have been lifted by now, must be *put on top level*. As this changes the structure of the abstract syntax tree, this step is realized by calling the procedure `AddASTsToDefSeq` of the data encapsulation of the abstract syntax tree (`EveAST`). In the current implementation all the lifted functions are added in front of the whole definition sequence that defines the EVE program.

After having executed these steps, the program consists of a collection of functions definitions that are all on top level (so there are no nested function definitions any more). None of these functions has free variables any more. These characteristics of the program are essential for the last pass in the compiler.

4.3.3 Further Transformations on the Abstract Syntax Tree

Apart from the λ-lifting, which has been described in detail in the previous section, there are also some further transformations performed on the abstract syntax tree that has been produced by the first pass. These transformations can be described as follows:

- One transformation *replaces all calls to predefined* comparison *functions*, to arithmetical and boolean operation functions and to the predefined list constructor function by the node of the abstract syntax tree that represents that function. Figure 4.2 shows

$$\left(\begin{array}{lll} \textbf{CASE } E & \textbf{OF} & \{\ v_1 \ :\ T_1\ \}\ E_1 \\ & & \{\ v_2 \ :\ T_2\ \}\ E_2 \\ & & \quad\vdots \\ & & \{\ v_n \ :\ T_n\ \}\ E_n \\ & \textbf{ELSE} & E_{n+1} \end{array}\right)$$

$$\downarrow$$

$$\left(\begin{array}{ll} \{\ x\ ==\ E\ ; & \\ \ \ h\ ==\ HEAD(x); & \\ \ \ t\ ==\ TAIL(x)\ \} & \\ \textbf{IF } h\ =\ tag(v_1) & \textbf{THEN } \{\ v_1\ ==\ t\ \}\ E_1 \\ \textbf{ELSIF } h\ =\ tag(v_2) & \textbf{THEN } \{\ v_2\ ==\ t\ \}\ E_2 \\ & \quad\quad\vdots \\ \textbf{ELSIF } h\ =\ tag(v_n) & \textbf{THEN } \{\ v_n\ ==\ t\ \}\ E_n \\ & \textbf{ELSE } E_{n+1} \end{array}\right)$$

x, h, t are new unique identifiers
$tag(v)$ yields the number of the variant with the name v.

Figure 4.3: Translation of a CASE Expression into an IF-THEN-ELSE expression

how a call of the predefined integer addition function `ADD` is in the first pass translated into an abstract syntax tree. In the second pass this tree is transformed into a tree without a function call but with a predefined function that also represents integer addition. This transformation process is similar for the following predefined functions:

- `ADD`, `SUB`, `MUL`, `DIV`, `MOD`,
- `NEG`, `ABS`, `ODD`,
- `AND`, `OR`, `NOT`,
- `*LT`, `*LE`, `*EQ`, `*NE`, `*GE`, `*GT`,
- `CONS`.

- In another transformation process *all case expressions are translated into explicit IF-THEN-ELSE constructs*. Figure 4.3 shows this in the form of a rewrite rule on the EVE source code. The basic idea behind this transformation is based on the fact that each object of a union type carries in its first field a tag denoting its actual type. Therefore, a CASE expression can be translated into an IF-THEN-ELSE expression that chooses the right branch by checking for the value in the first field of this object.

  After this transformation, the abstract syntax tree does not contain any CASE expressions any more. Therefore, these expressions are not covered in the next section,

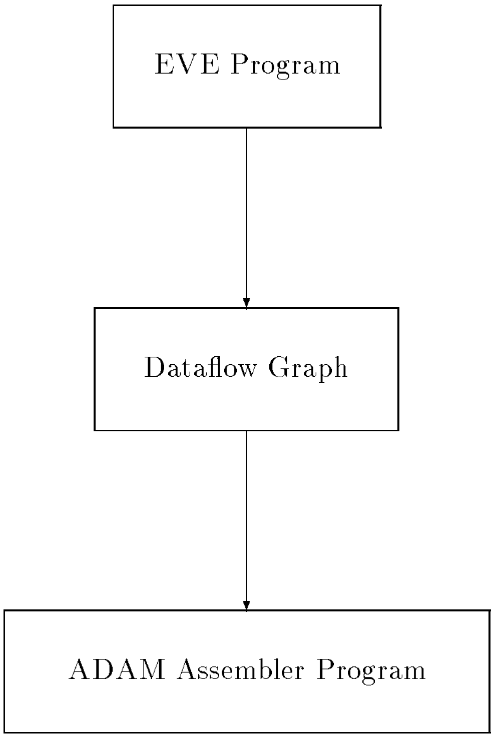


Figure 4.4: Description of the Code Generation

where the code generation for the various constructs is described.

- In the last transformation a *free variable analysis* is performed and as a result of this analysis to each node in the abstract syntax tree the set of free variables in the subtree, starting from that node, is attached. This transformation is done by the procedure `AttachFreeVars` in the data encapsulation of abstract syntax trees (`EveAST`). This information about which variables are free in a subtree is needed at each node in the tree for the code generation step as we will see in the next section.

4.4 Third Pass

In this section we give an informal description of the code-generation step in the EVE compiler. However, a complete formal description of this step in the form of a denotational semantics can be found in Appendix B. Hence, the reader may refer to this appendix to look up some details that are omitted in this section in order to concentrate on the main features of the translation.

The central idea in our approach for an automatic parallelization of functional programs is to simulate the dataflow behaviour of a functional program on a control-driven machine that has special constructs for the creation and for the synchronization of parallel threads of control. Therefore, dataflow graphs play a crucial role in the translation process although there is no explicit data structure of a dataflow graph used in the compiler. Moreover, these dataflow graphs show the inherent parallelism that is contained in the EVE language constructs. Therefore, the description of the translation of the various EVE language constructs into ADAM assembler code is split into two parts:

1. First, the translation of the EVE constructs into dataflow graphs will be discussed. This step already shows which parts of the computation may be performed in parallel.
2. Afterwards, it will be shown how these dataflow graphs can be translated into ADAM assembler code that simulates the dataflow behaviour of these graphs.

This idea of using two steps in presenting the code-generation step of the EVE compiler can be also seen from Figure 4.4 where the overall structure of the code generation is presented graphically.

Of course, in the compiler itself there is only one step performed, namely a direct translation of EVE constructs into ADAM assembler code. But in order to understand the motivation for using the translation schemes that will be presented these dataflow graphs are very helpful.

The main advantage of this approach is that one can exploit the inherent parallelism of the dataflow model without losing the efficiency of the controlflow model. As it has been described in Section 3.1 a pure dataflow model needs a lot of copying of result values. However, by using the concept of registers from the controlflow model many of these copying instructions become superfluous. Another advantage of this approach is that some methods from compiler design that have been developed for the optimization of imperative programs can now also be applied to the code that is produced by the EVE compiler.

4.4.1 From EVE Programs to Dataflow Graphs

The structure and the semantics of dataflow graphs have already been discussed in Section 3.1.1. From the description given there one could get the impression that it suffices to have all free variables of an expression as the input of a dataflow graph and the result of evaluating the expression as the only output of the graph. However, when trying to implement such graphs one gets into some problems:

- If there are expressions without any free variables (e.g. constants) the corresponding dataflow graphs should deliver a result whenever it is needed by one of the succeeding nodes. To implement this concept of producing a token "on demand", an additional input arc, the *trigger* arc, is introduced. The arrival of a token at this arc indicates that the result of the expression is really needed and should therefore be computed. This trigger arc is also very important for implementing non-strict functions as it allows to start the computation of a function before all arguments have arrived, by sending such a trigger token to the function.
- Another problem stems from the presence of side effect nodes in the dataflow graph. In general, all operations on compound data structures are realized via side effects. Some of these nodes have no actual result (e.g. the node that writes a value into a special field of a tuple). However, it must be possible to tell a succeeding node that such a side-effect node has fired. For that purpose an additional output arc, the *signal* arc, is

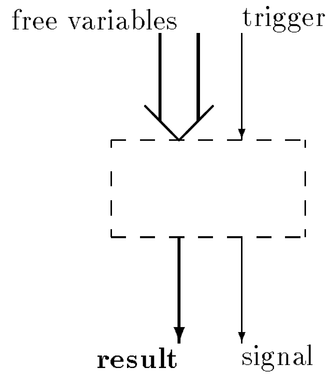


Figure 4.5: The Interface of a Dataflow Graph

introduced. Now, the following invariant holds in the entire dataflow graph: Whenever tokens have appeared on each of the output arcs it is guaranteed that there are no tokens in the dataflow graph left any more. This property is often called *self-cleaning* property of dataflow graphs.

Figure 4.5 shows the resulting general interface of an arbitrary dataflow graph. In this graph the arcs on which tokens flow that really carry values are drawn thicker than arcs where arriving tokens do not have well defined values. Note that tokens that arrive at the trigger arc as well as tokens that are produced at the signal arc need not carry any values as just the arrival or production of a token on the corresponding arc represents all the information one wants to get from these tokens. For a more detailed discussion of triggers and signals see [Traub, 1986][Chapter 6].

In the following sections for each EVE language construct the corresponding dataflow graph is presented. These graphs are presented in a bottom up fashion. Hence, the simplest graphs are presented at the beginning and the more complex graphs are described at the end of this section.

Constants

One of the simplest EVE constructs, namely a constant, is shown in Figure 4.6. Note that due to the description of the interface of each dataflow graph in Figure 4.5, this dataflow graph has only one input arc, the trigger arc. Whenever a token arrives at that input arc an output token is produced. This output token then has to be copied as every dataflow graph must have two output arcs. The token that arrives at the result output carries the value c. However, the value of the token at the signal output is undefined.

Variable

The dataflow graph of a variable in Figure 4.7 is even simpler than that of a constant. All that has to be done is to forward the token on the input arc that corresponds to the

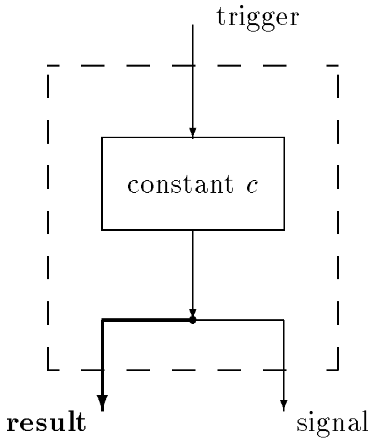


Figure 4.6: Dataflow Graph of a Constant: c

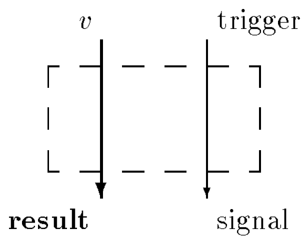


Figure 4.7: Dataflow Graph of a Variable: v

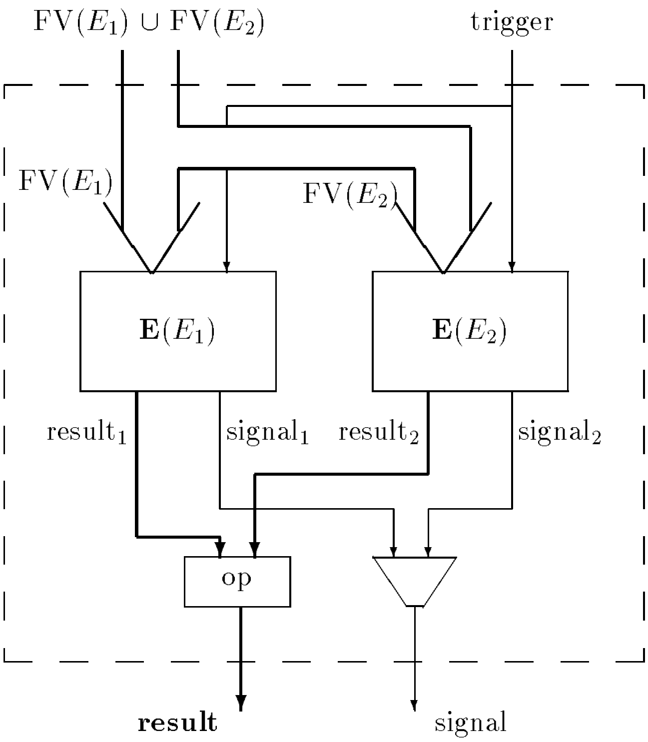


Figure 4.8: Dataflow Graph of a Binary Operation: E_1 op E_2

variable to the result output. The incoming trigger token can be directly forwarded to the signal output of the dataflow graph. There is no synchronization between the two input arcs necessary as the appearance of a token at the output arc does not directly imply that there are no tokens in the dataflow graph left any more. Moreover, only when at all output arcs tokens have arrived we can conclude that the dataflow graph has been 'cleaned up'. Therefore, it is very well possible that the signal output arrives even before the actual result has been computed[9].

Binary Operations

The dataflow graph in Figure 4.8 shows how a binary operation is performed on two expressions E_1 and E_2. As the incoming trigger token is immediately split into two successor tokens, both expressions are evaluated in parallel. As for any dataflow graph the set of the free variables has to be split into the sets of free variables of the subexpressions. As soon as both results of the subexpressions have been computed the operation is applied on these results. Note that the semantics of dataflow graphs requires both inputs the 'op' node to be present before this node can fire. The resulting token is then also the result of the whole binary operation. However, the signal outputs of the subexpressions E_1 and E_2 also have to

[9]This way of defining the semantics of the signal output minimizes the necessary synchronization and therefore maximizes the parallelism in the dataflow graph.

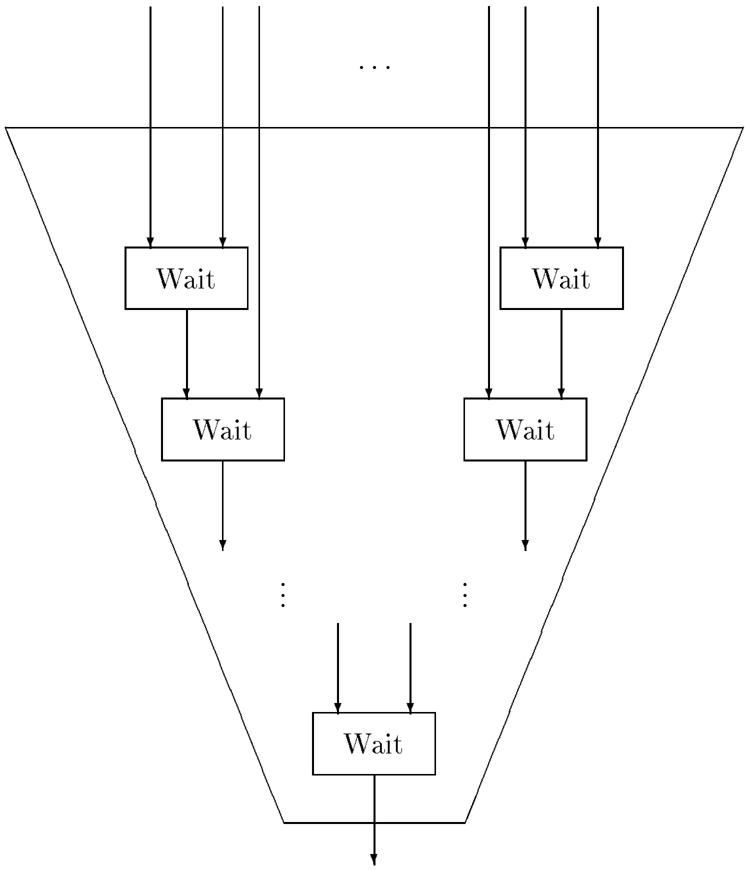


Figure 4.9: Internal Structure of a Signal Tree

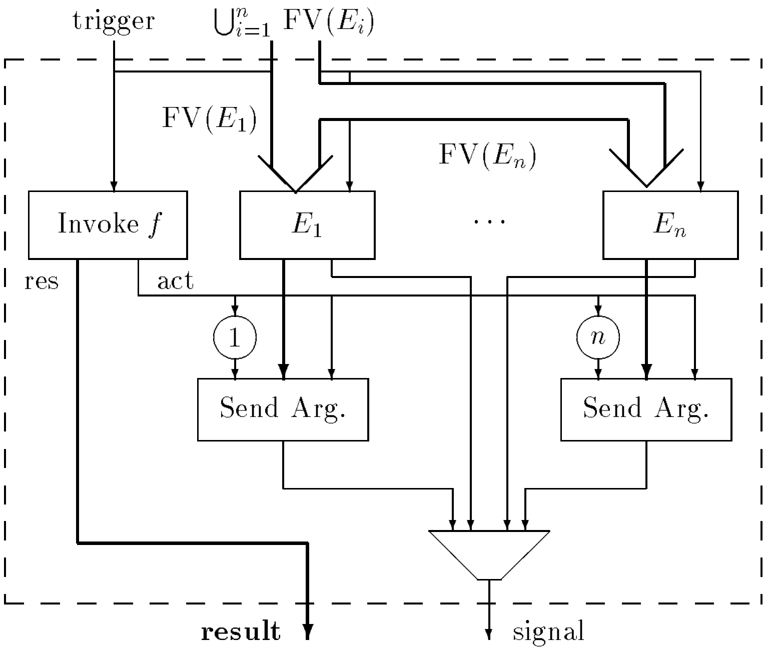


Figure 4.10: Dataflow Graph of a Function Call: $f[E_1, \ldots, E_n]$

be resolved. This is done by connecting these two arcs with a so called *signal tree*. Figure 4.9 shows the internal structure of a signal tree. From this figure one can see that a signal tree simply waits for all of its inputs, there may be arbitrary many, and produces an output token when all input tokens have arrived. The value of the output token is undefined. This reflects the fact that a signal tree is only used for the synchronization of the signal tokens, which themselves do not carry well defined values.

Function Call

The function call is a very interesting construct from the parallelism point of view. Obviously, all arguments of the function call may be evaluated in parallel. This exploits the horizontal parallelism, which is also available in strict programming languages. However, as EVE is a non-strict programming language the computation of the function can be started in parallel with the computation of the arguments. This is usually called vertical parallelism. When in the evaluation of the function an argument is needed that has not been computed yet, the evaluation of the function is suspended until that argument is available. This synchronization is realized by using non-strict data structures as they have been described in Section 3.1.

The $n+1$ parallel threads in the function call can be seen in Figure 4.10 as the trigger token is split into $n+1$ successor tokens. The first of these successor tokens starts the invocation of the function, the other n tokens start the evaluation of the n arguments. The 'invoke' node in this dataflow graph is of special interest as it has to start the computation of the function itself. This is done by performing a side effect. To be more exact, this node performs the following actions:

- It has to initiate the allocation of an activation record for the called function[10].
- It has to initiate the evaluation of the function itself.

Corresponding to these two tasks the 'invoke' node has two output arcs:

- The 'act' output arc yields a token that carries a pointer to the allocated activation record as its value. This token is needed for sending the arguments to the correct function activation.
- The 'res' output arc yields a token that carries the result of the computation of the function. Therefore, this token can be directly forwarded to the result of the dataflow graph.

Due to the non-strict semantics of user defined functions in EVE, the invocation of the function and the sending of the arguments have been split into independent parts. However, for the sending of the arguments the pointer to the activation record of that function must be present. Therefore, the 'SendArg' nodes, one node for each argument, have two input tokens and a constant input token. If all tokens are present the node performs a side-effect by sending the argument to the correct function activation. As this side-effect node has no actual result the only output arc is a signal output. All these signal arcs and the signal arcs from the argument expressions are finally synchronized in a signal tree to yield the signal output for the whole dataflow graph.

Conditional Expression

The conditional expression is probably the most complicated expression regarding the translation into a dataflow graph. This can already be seen from the complexity of the resulting dataflow graphs in the Figures 4.11 and 4.12. To make the structure of this dataflow graph clearer it has been split into two parts: a head and a body.

Figure 4.11 shows the dataflow graph of the head of the conditional expression

$$\texttt{IF } E_1 \texttt{ THEN } E_2 \texttt{ ELSE } E_3.$$

There, first the expression E_1 has to be evaluated. Based on the result of this evaluation, which arrives at the result_{E_1} arc, the free variables of the two body expressions, the trigger input and the signal of this expression are either forwarded to the THEN branch (E_1) or to the ELSE branch (E_2). Therefore, it is guaranteed that all input tokens of the switch node are forwarded to the same branch. After having evaluated the appropriate branch the results of both branches have to be merged together as it is not possible to decide at compile time which branch has been chosen. However, it is guaranteed that only from one of the two branches a result is returned. The same merging has to be done with the signal outputs of the two branches.

As the THEN and the ELSE branch are essentially the same, Figure 4.12 shows a dataflow graph that is parameterized with the number i, where $i = 2$ stands for the THEN branch and $i = 3$ stands for the ELSE branch. In this dataflow graph two items have to be emphasized:

[10] The actual allocation has to be done by the memory management unit.

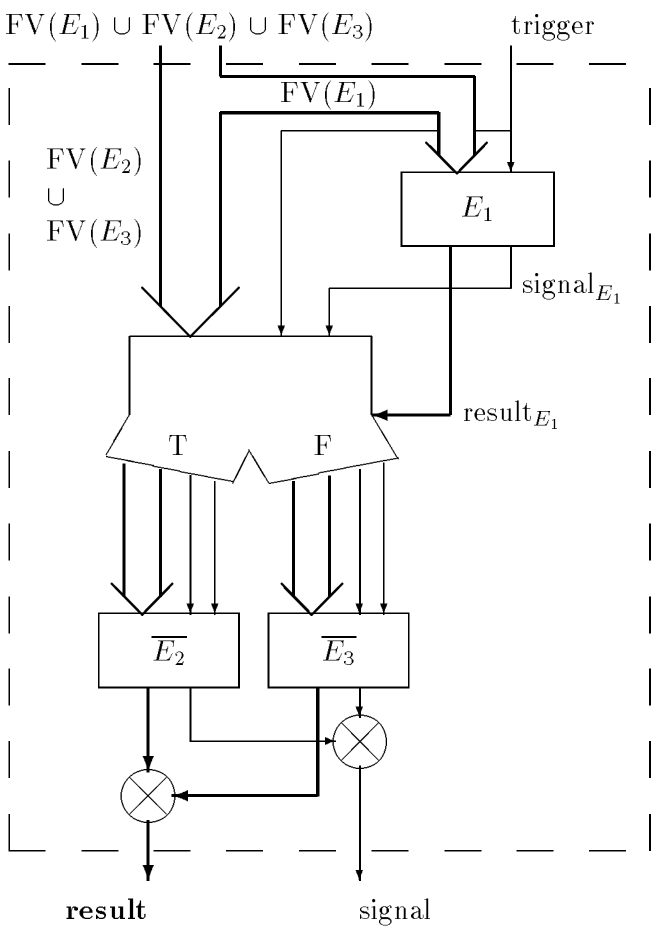


Figure 4.11: Dataflow Graph of the Head of a Conditional Expression

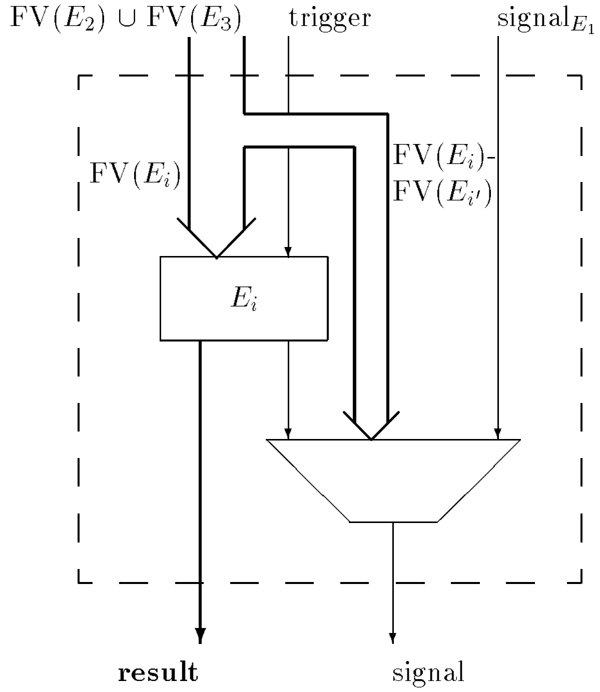


Figure 4.12: Dataflow Graph of the Body of a Conditional Expression

- Not all free variables that are the inputs of this dataflow graph are needed in the expression. Those free variables that are only needed in the other branch have to bypass the expression and can be directly forwarded to the signal tree.

- The trigger input of this dataflow graph must be the trigger of the whole conditional expression. It is not possible to take the signal of the expression E_1 as the trigger as E_1 might not terminate but nevertheless yield a result. In this case the whole conditional expression would not terminate if we used 'signal_{E_1}' as the trigger for the expressions E_2 and E_3. But this would violate the semantics of the conditional expression. As 'signalE_1' can't be used as a trigger it must be directly forwarded to the signal tree in both branches.

The central insight from the parallelism point of view is here, that the result of the expression E_1 must have been computed before the computation in the appropriate branch can be started. This is one of the central synchronization points in an EVE program. Therefore, one should avoid conditional expressions in which the evaluation of the head takes most of the time. Wherever possible the expensive computations should be pushed into the two branches and only cheap computations should be performed in the head. This is especially important when we have a recursive function where a conditional expression is necessary to detect the base case of the recursion. If the head contains expensive computations, there won't be many overlapping computations between the different levels in the recursion. However, if the the computation of the head is quite cheap many different levels of recursion can be evaluated in parallel. For a more detailed description of this behaviour see Section 5.2.

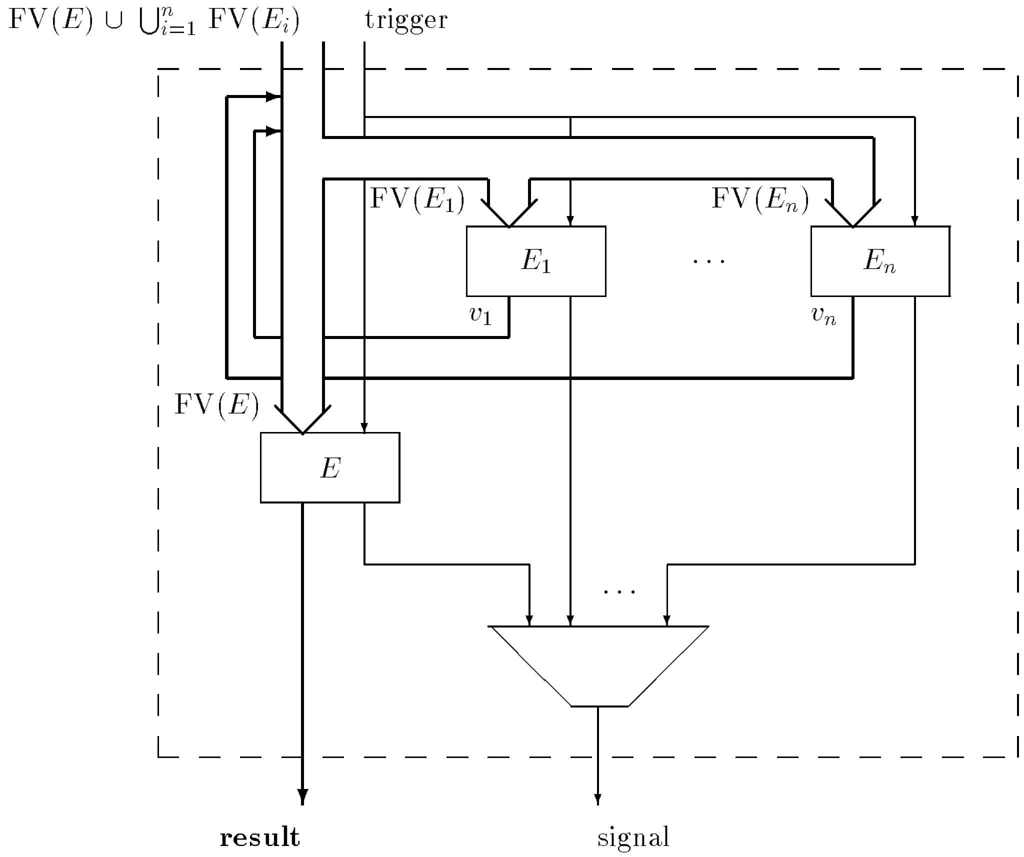


Figure 4.13: Dataflow Graph of a Local Definition: $\{v_1 == E_1, \ldots, v_n == E_n\}E$

Local Definitions

Figure 4.13 shows the dataflow graph of an expression, where the objects $v_1, \ldots, v_n$ are defined locally to that expression[11]. Examining the splitting of the trigger token we see, that all locally defined objects and the main expression itself are evaluated in parallel[12]. As the object v_i may refer to objects v_j for $j < i$ it would be wrong to just forward the object to the main expression. Instead, it has to be put into the stream of free variables that enters the dataflow graph. The result of the main expression can be directly forwarded to the result of the whole expression. The signals of all expressions have to be synchronized in a signal tree.

As this dataflow graph shows, all local objects will be evaluated in parallel even if the some of the objects are not needed at all. It might even happen that the evaluation of some local object does not terminate although a result is returned by the main expression. In this

[11] Note that we need not consider local function definitions as they have already been lifted on top level by the second pass.

[12] As in the function call, the computation of the main expression is automatically suspended when an object is needed that has not been evaluated yet.

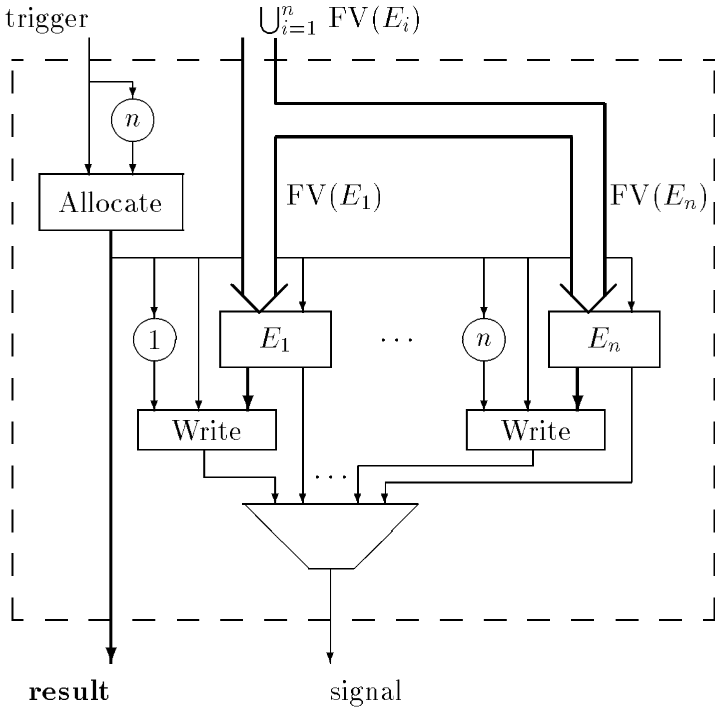


Figure 4.14: Dataflow Graph of a Tuple Construction: $[E_1, \ldots, E_n]$

case a token carrying the result leaves the dataflow graph but a signal output will never be produced.

Tuple Construction

The dataflow graph of the tuple construction in Figure 4.14 is quite similar to the dataflow graph of the function call. The main difference is the fact that the allocation of a tuple is done sequentially before the expressions of the various components are evaluated. However, as it has been described in Section 3.1 the allocation of some memory is a split-phase instruction and therefore tolerates long memory latencies. Therefore, this part has been implemented sequentially. The 'Allocate' node performs a side-effect, as it causes the memory unit to allocate some memory. The result of this node is a token carrying a pointer to the allocated memory. As in EVE data structures are non-strict, this token is immediately forwarded to the result of the whole dataflow graph. Hence, another instruction may use this data structure even before all of the arguments have been computed. This is possible since the data structures are implemented in such a way that with each field of the data structure a full/empty bit is associated. This corresponds to the idea of I-structures that are described in [Traub, 1986] and [Arvind *et al.*, 1987]. For a more detailed discussion of this concept see Section 3.1.

In the other n parallel threads the values of the expressions are evaluated. As soon as one value is available it is written into the correct field of the tuple. This is done in the node

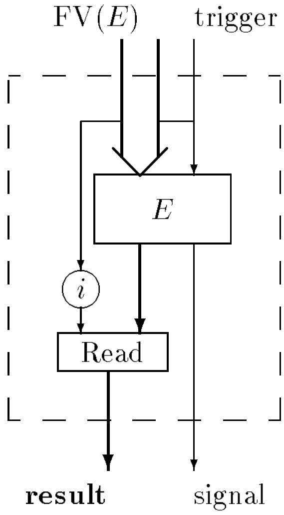


Figure 4.15: Dataflow Graph of a Tuple Selection: $\pi_i\ E$

'Write'. This node actually performs a side effect[13] and has as result just a signal output arc. To keep the self-cleaning property all these signal arcs and the signal arcs of the expressions have to be synchronized in a signal tree.

Tuple Selection

In Figure 4.15 a dataflow graph is presented that allows to select a special field of a tuple. In the EVE language this is described by using a selector of the form *Tuple.Field*. However, in the abstract syntax tree this has been transformed into a selection of the i-th field of that tuple. In the denotational semantics this is described by the construct $\pi_i\ E$.

The interesting part in this dataflow graph is the 'Read' node that takes an integer value i and a pointer to a tuple T and initiates the selection of the i-th field of T by the memory management unit. This side-effect node then produces a token that carries the value of the i-th field of T.

Function Application

After having discussed the translation of the basic EVE expression constructs into dataflow graphs in the previous sections, we will now cover the translation of two advanced constructs that deal with higher order functions.

In Section 4.4.1 only the case of first order functions was covered. However, as in EVE higher order functions are allowed it is possible to apply an expression rather than a simple name of a function to some arguments. The translation of such a function application into a dataflow graph is shown in Figure 4.16.

[13] All reading and writing operations of data structures are performed by the memory management unit. The corresponding nodes in the dataflow graph just initiate the operation of the memory management unit.

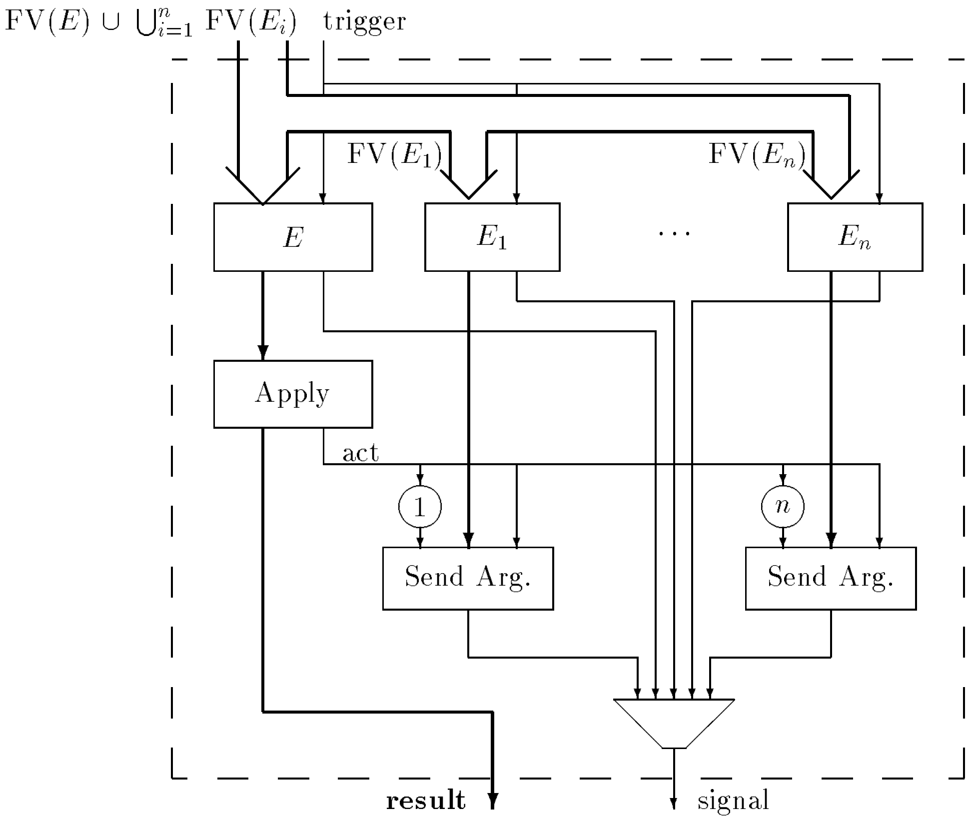


Figure 4.16: Dataflow Graph of a Function Application: $E[E_1, \ldots, E_n]$

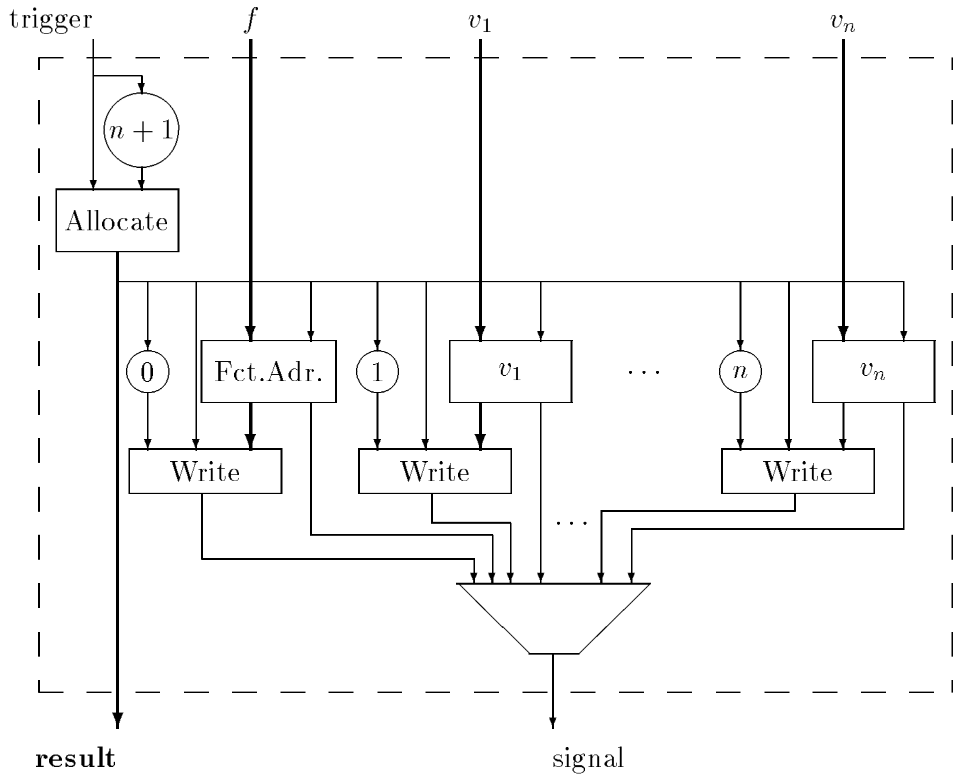


Figure 4.17: Dataflow Graph of a Curried Expression: $< f, v_1, \ldots, v_n >$

The overall structure is similar to that of the function call. However, in this case the function is the result of an expression E. The 'Apply' node has the same tasks as the 'Invoke' node but it takes the function that has to be applied as an input.

Currying

Although an explicit currying of functions is not possible in EVE, the abstract syntax tree as an intermediate language offers the possibility to represent curried expressions. In the current version of the compiler this is only exploited for higher order functions (e.g. functions that are parameters to other functions) that are represented as curried expressions with 0 curried variables.

The dataflow graph in Figure 4.17 is actually just a special case of a tuple construction. However, as it is a special case some parts can be implemented more efficiently than in a general tuple construction. For example no general expressions have to be evaluated as the components of the tuple that is created are just the curried variables. In the first component of the tuple the address of the curried function and the number of the curried variables are encoded. This encoding is performed by the 'Fct.Adr.' node.

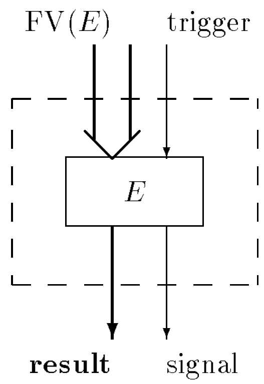


Figure 4.18: Dataflow Graph of a Simple Definition: $v == E$

Simple Definitions

After having discussed the translation of EVE expressions into dataflow graphs it now remains to describe the translation of EVE definitions into dataflow graphs.

As it can be seen from Figure 4.18, a simple definition has the same interface as the expression that defines a new object. Therefore, in the dataflow graph the input arcs are directly forwarded to the input arcs of the expression and the output arcs of the expression are directly forwarded to the output arcs of the simple definition. The only reason to introduce such a simple expression is the ability of giving a name to some expression. This directly reflects the semantics of a variable binding in functional languages (see Section 2.1).

Function Definitions

Contrary to simple definitions, function definitions have no explicit input or output arcs. Moreover, when the function is called dynamic links to this dataflow graph are established (this is done in the nodes 'Invoke' and 'SendArg' that have been introduced in Section 4.4.1). And when the result has been computed a dynamic link to the corresponding 'Invoke' node is established such that this node can emit a token carrying the result of the function. To denote this notion of dynamic links the arcs for the trigger token and for the formal parameters are drawn from the left and are not entering the dataflow graph but they have their origin in the border of the graph. The same principle has been used to describe the result of the function.

The two interesting nodes in this construct are:

- The 'Return' node that establishes the dynamic link from this function definition to the 'Invoke' node that called this function. Thereby, the resulting value is transmitted to the caller of the function.
- The 'Terminate' node destroys the incoming token. As a side-effect it tells the memory management unit that the activation record for this function activation can be discarded.

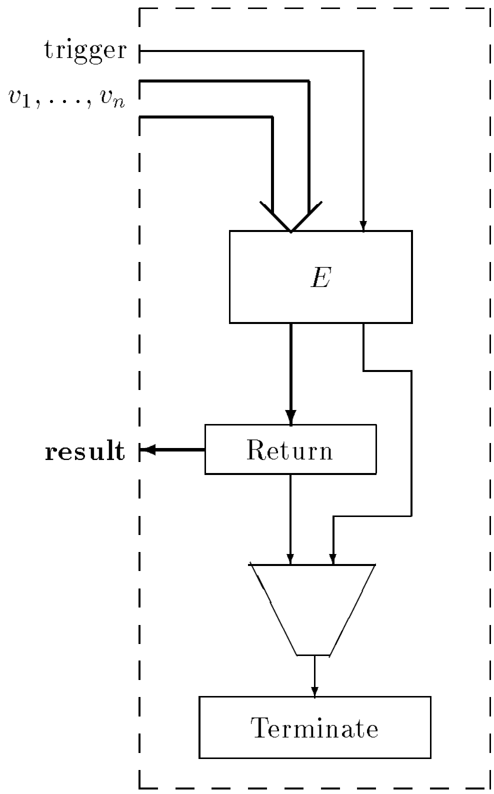


Figure 4.19: Dataflow Graph of a Function Definition: $f[v_1, \ldots, v_n] == E$

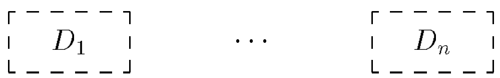


Figure 4.20: Dataflow Graph of a Definition Sequence (Top Level): $D_1, \ldots, D_n$

Definition Sequences

Here one has to distinguish between definition sequences on top level, where only functions and the constant `MAIN` are defined and nested definition sequences that contain only simple definitions. The latter case has already been handled in the discussion of local definitions[14]. In the former case we just have a collection of unconnected function definitions as they are described in the previous section. Therefore, the dataflow graph looks like in Figure 4.20.

Program

As one can see from the syntax definition of EVE in Section 3.2.13 a program is just a special definition sequence. Therefore, the dataflow graph for a definition sequence can also be used here. However, the only constant on top level, namely `MAIN`, has to be handled in a special way. The compiler does this by extracting the definition for `MAIN` out of the program. The rest of the program can then be translated as it is described for definition sequences of function definitions. This approach is useful as the compiler has to produce two different files anyway:

- A program file containing a collection of function definitions. This file will be produced from the program without the constant `MAIN`.
- An expression file containing the code for the top level expression. This file will be produced by applying the usual translation of an expression on the body of the constant `MAIN`.

4.4.2 From Dataflow Graphs to ADAM Assembler Code

As our approach to the automatic parallelization of functional programming languages does not rely on special hardware support but is based on conventional hardware, the dataflow graphs have to be translated into the assembler code of an abstract machine, namely the ADAM machine. A description of the ADAM machine has been given in Section 3.1. A summary of the ADAM instruction set with a discussion of the various instructions can be found in Appendix C.

As it has already been mentioned, the ADAM assembler code is essentially a conventional two address code with extensions for creating and synchronizing parallel threads of control.

[14] The reader should be aware that local definitions are the only possibility where nested definition sequences might occur.

Therefore, in this step we have to describe the translation of the dataflow model into the controlflow model. As this is obviously more complex than a translation that remains within one model we will first discuss the general correspondencies of the entities in the dataflow and the controlflow model. Afterwards, we can, based on these correspondencies, give the translations of the dataflow graphs of Section 4.4.1 into ADAM assembler code.

In the description of this translation it turns out that the handling of the free variables forms a quite big part without reflecting any crucial concepts of a parallel implementation. In order to concentrate on the important parts of the translation this handling of free variables is usually omitted in the presentation of the ADAM assembler code. However, this is only true for the description in this chapter. In Appendix B a denotational semantics of the whole code generation process in the EVE compiler is given. There all details of the translation are covered.

A Comparison between the Dataflow Model and the Controlflow Model

The translation of dataflow graphs into ADAM assembler code requires a study of the correspondencies between the central entities in the dataflow model and in the controlflow model. By this it is then possible to simulate a dataflow behaviour on a control driven machine.

Now, we will discuss for each of the important entities of the dataflow model its correspondence with entities in the controlflow model:

- As we want to get a parallel assembler program the constructs dealing with parallelism are most important for us. The *creation of parallelism*, that is realized by copying a token (especially the trigger token), into several successor tokens, is done in the controlflow model with an instruction that splits the thread of control into two independent threads (the SPLIT instruction in the ADAM code). The *synchronization of parallelism*, that is realized by waiting for two input tokens in order to produce one output token in the dataflow graph, is done in the controlflow model by a waiting instruction (WAIT in ADAM code) that merges two threads of control into one thread.
- The *connections between the operations*, that are realized by the arcs between nodes in the dataflow graph, are established in the controlflow model by explicit jumps. ADAM instructions usually have a field that states the relative address of the next instruction to be performed. This reflects the situation in the dataflow model. The straightforward translation of a dataflow graph into a dataflow program that is presented in Section 3.1.1 also shows the typical structure of the instructions containing the address of the successor.
- Due to this concept of 'translating arcs into jumps' the following correspondencies become obvious:
  - The *names of the arcs* in the dataflow graph become *labels* in the controlflow program.
  - The *nodes* in the dataflow graph become *(sequences of) instructions* in the controlflow program.

- In the previous section it has been stated that some nodes in the dataflow graph perform side effects. The corresponding ADAM assembler instructions are characterized by the fact that the result of the side-effect instruction will appear at the next instruction in the program. Therefore, this instruction will usually be a RECEIVE that puts the value it gets in the register that is the first argument of this instruction.

Abstract Data Types for the Code Generation

Before we can discuss the code generation itself, the abstract data types that are needed for the creation of ADAM assembler code have to be described. The following data types are needed in the code generation:

- **Variables.** This data type contains all possible variables in an ADAM assembler program. The set of all variables can be expressed as a language of the grammar given in Section 3.2.13 with *Ident* as starting symbol. The only operation that must be performed on these objects is the test for equality.
- **Flags for checking compound objects.** This data type is needed to decide whether an ADAM object is an integer value or a pointer to a compound data structure. Such a distinction is necessary to guarantee that the garbage collection operates in the intended way.
- **Registers.** The register data type represents registers of the ADAM machine. However, they need not directly correspond to the registers of the underlying hardware. In the ADAM machine arbitrarily many registers are available. Therefore, there is no explicit procedure for "dropping" a once fetched register. Moreover, all registers are local to a function and therefore all registers of a function can be dropped at the end of the function. The most important operation that is performed on this data type is to get a unique register out of the pool of available registers. In the following, registers will usually be denoted by $r, r_1, \ldots$.
- **Labels.** The objects of this data type are needed for the jump instructions. As with the registers, an operation is needed that gets a unique label out of the pool of available labels. There are arbitrarily many labels, which are local within a function. Labels will usually be denoted by $l, l_1, \ldots$.
- **Events.** Events are needed by WAIT instructions to synchronize parallel threads of control. When the first WAIT on an event is applied the thread of control is suspended. As soon as a second WAIT instruction is performed on the same event one thread continues with the successor instruction. By this construction two threads of control are joined into one thread. As with registers and labels the most important operation on this data type is the selection of a unique event. There are arbitrarily many events, which are local within a function. Events will usually be denoted by $e, e_1, \ldots$.
- ADAM **Assembler Code.** See Appendix C for a description of the ADAM instruction set.
- **Environment.** The environment is carried through the whole compilation process. At the beginning of the compilation it is empty. For each of the variables that are currently in scope, the environment contains

trigger :	CONST	$r, +2$	; Get constant into register r
	c'		; Value of the constant
	SPLIT	*result*, *signal*	; 2 output labels required

Figure 4.21: ADAM Assembler Code of a Constant: c

- the register that carries the value of that variable,
- the label that announces the availability of this register,
- a flag that indicates whether this variable contains an integer value or a pointer to a compound data structure.

The operations that are performed on an environment are

- the binding of a variable to a triple of register, label and flag,
- the binding of a variable to a new label,
- a selection operation for each of the three components of an environment.

- **State.** The overall state consists of
  - the state of the registers,
  - the state of the labels,
  - the state of the events.

  The states of registers, labels and events contain the information that guarantees that every new register, label or event is unique. Therefore, this state must also be carried through the whole compilation process.

A formal description of these abstract data types and of the properties that must hold for these structures can be found in the denotational semantics of the code generation in Appendix B.

Constants

As for all EVE constructs, the input and the output of this construct are:

- **Input:** *trigger*.
- **Output:** *result*, *signal*.

Figure 4.21 shows the assembler code that has to be created for a constant. The code sequence is entered by a jump to the label *trigger*. This reflects to the dataflow point of view where the arrival of a trigger token starts the computation of the expression. Depending on the type of the constant, c' gets one of the following values:

trigger :	SPLIT	l_{E_1}, l_{E_2}	; 2 parallel pathes
	$\mathbf{C}_{E_1}$		; evaluate 1^{st} subexpression
	$\mathbf{C}_{E_2}$		; evaluate 2^{nd} subexpression
$result_{E_1}$:	WAIT	$e_1, +2$	; synchronize the results
$result_{E_2}$:	WAIT	$e_1, +1$	; of both subexpressions
	COPYI	$r, r_{E_1}, +1$	; r will contain the result
	op	$r, r_{E_2}, result$	; perform binary operation
$signal_{E_1}$:	WAIT	$e_2, signal$	; synchronize the signals
$signal_{E_2}$:	WAIT	$e_2, signal$	; of both subexpressions

Figure 4.22: ADAM Assembler Code of a Binary Operation: E_1 op E_2

- If the constant is of the form `numberN` c (i.e. it is an integer constant) then the value of c' is c.
- If the constant is of the form `charN` c (i.e. it is a character constant) then the value of c' is the ASCII value of c denoted by `ORD`(c).
- If the constant is `TRUE` (i.e. it is a boolean constant) then the value of c' is 1.
- If the constant is `FALSE` (i.e. it is a boolean constant) then the value of c' is 0.

The CONST instruction takes the value of the next line and puts it into the register r. As it can be seen from the dataflow graph in Figure 4.6 the result of this operation has to be split into two threads (two tokens from the dataflow point of view). These two threads are the output threads of this construct.

Variables

As the dataflow graph in Figure 4.7 already shows there need not be any code generated for a variable at all. However, somehow it must be stated where to get the value of this variable from (this means that the correct arc in the dataflow graph has to be chosen as input arc). This is done by carrying an environment through the whole compilation, in which the variables are bound to a triple consisting of a register, a label and a flag.

Binary Operations

The first line in Figure 4.22 already shows that the two expressions of the binary operation are evaluated in parallel. The interface of the subexpressions $\mathbf{C}_{E_i}$ (where $i = 1, 2$) is described by the following input/output definition:

- **Input:** l_{E_i}.

	$\mathbf{C}_{split,trigger}$		; Feed trigger to fct and
			; to each argument
invoke :	FASTCALL	$f, +1$	; Invoke function
act :	RECEIVE	$r_{act}, send$	; Fct activation arrives
res :	RECEIVE	$r, result$	; Result of fct arrives
	$\mathbf{C}_{split,send}$		; One send for
			; each argument
	$\mathbf{C}_{E_i}$		; $i = 1, \ldots, n$
$result_{E_i}$:	WAIT	$e_i, +2$	; Synchronize result of
$send_{E_i}$:	WAIT	$e_i, +1$	; arg. with fct. invocation
	SEND	r_{E_i}, r_{act}, i	; send ith arg. to fct.
	JUMP	arg_{E_i}	
	$\mathbf{C}_{SignalTree}$		

Figure 4.23: ADAM Assembler Code of a Function Call: $f[E_1, \ldots, E_n]$

- **Output:** $result_{E_i}, signal_{E_i}$.

After the computation of the values of the subexpressions the two results have to be synchronized with two WAIT instruction that refer to the same event e_1. As soon as both results are available, the computation is continued with a COPYI instruction. This instruction is necessary as all binary arithmetical and boolean instructions in the ADAM instruction set put the result of the operation in the first argument thereby altering the value of the argument. The result value of this operation is immediately made available to succeeding parts of the program by performing a jump to *result*. The last two lines in the code represent the signal tree with two inputs. From now on the code for the signal trees will be abbreviated by a special code block.

One important item of this instruction sequence is that the binary operation has to be 'guarded' with a WAIT instruction. In the dataflow model this was incorporated in the dataflow semantics of the firing rules. But in the controlflow model all these synchronizations must be expressed explicitly.

Function Call

Due to the non-strict semantics of user defined functions in EVE, the evaluation of the function can be started in parallel with the evaluation of the function itself. Therefore, the code sequence in Figure 4.23 starts with a splitting of the controlflow into $n+1$ parallel threads. This is done in the code sequence $\mathbf{C}_{split,trigger}$, which has the following interface:

- **Input:** *t*rigger.

- **Output:** *invoke*, $trigger_{E_1}$, ..., $trigger_{E_n}$.

In the first parallel thread, namely in the *invoke* thread, the given function is immediately called. As this calling of a function is performed via a side effect the answer will be received in the next line. There the pointer to the activation record will be stored in the register r_{act}. However, as the corresponding dataflow graph shows, the invocation of a function has two results. The second result, namely the actual result of evaluating the function, is stored in register r and is immediately made available by the jump to *result*.

As the pointer to the activation record of the function is needed for the sending of the arguments, the thread from the first result of the FASTCALL instruction (denoted by *send*) is split into n parallel threads (one for each argument). This splitting is performed in the code sequence $\mathbf{C}_{split,send}$, which has the following interface:

- **Input:** *send*.
- **Output:** $send_{E_1}, \ldots, send_{E_n}$.

Each thread is synchronized with the result of the corresponding argument.

The interface for the argument expressions that are denoted by $\mathbf{C}_{E_i}$ (where $i = 1, \ldots, n$) looks as follows:

- **Input:** $trigger_{E_i}$.
- **Output:** $result_{E_i}, signal_{E_i}$.

After the synchronization of the result of the i-th argument expression with one of threads for sending arguments, this result is sent as the i-th argument to the function that is denoted by the activation record in r_{act}. As the SEND instruction performs a side effect its result arrives at the next line. However, this result is just a signal output. All the unresolved signal outputs in the code sequence are resolved in the signal tree that is denoted by $\mathbf{C}_{SignalTree}$. The interface of this signal tree is:

- **Input:** $signal_{E_i}, arg_{E_i}, i = 1, \ldots, n$.
- **Output:** *signal*.

Note that like the binary operation in the previous section the SEND instruction has to be guarded by a WAIT instruction to achieve a correct synchronization.

Conditional Expressions

Figure 4.24 shows the assembler code of the conditional expression:

$$\texttt{IF } E_1 \texttt{ THEN } E_{true} \texttt{ ELSE } E_{false}.$$

$trigger$:	SPLIT	$trigger_{E_1}, trigger_{E_{23}}$	; Split one path for body
	$\mathbf{C}_{E_1}$		; 1 path for head
	$\mathbf{C}_{split}$		; 1 split for each free var in $E_{2,3}$
$trigger_{E_{23}}$:	WAIT	$e, +2$	; Synchronize trigger with
$trigger_0$:	WAIT	$e, +1$	; result of E_1
	SWITCH	$r_{E_1}, l_{true,0}, l_{false,0}$	; Forward it to right branch
	COPYI	$r, r_{E_1}, result$	; error $\Rightarrow$ return error value
ly_i :	WAIT	$e_i, +2$	; $i = 0, \ldots, n$
$trigger_i$:	WAIT	$e_i, +1$	; Sync. as above
	SWITCH	$r_{E_1}, l_{true,i}, l_{false,i}$	; Forward var to right branch
	JUMP	$l_{error,i}$	; error $\Rightarrow$ jump to signal tree
	$\mathbf{C}_{E_{true}}$		; True branch
	$\mathbf{C}_{SignalTree_{E_{true}}}$		; Vars not used in this branch
	$\mathbf{C}_{E_{false}}$		; True branch
	$\mathbf{C}_{SignalTree_{E_{false}}}$		; Vars not used in this branch
	$\mathbf{C}_{SignalTree_{error}}$		; For the error case
$result_{E_{true}}$:	COPYI	$r, r_{E_{true}}, +1$	; r gets the result
	JUMP	$result$	; Leave code sequence
$result_{E_{false}}$:	COPYI	$r, r_{E_{false}}, +1$	; r gets the result
	JUMP	$result$	; Leave code sequence
$signal'_{E_{true}}$:	JUMP	$signal$	; Merge the signals
$signal'_{E_{false}}$:	JUMP	$signal$	; of both branches

Figure 4.24: ADAM Assembler Code of a Conditional Expression

As it can be seen from the dataflow graph for a conditional expression in Figures 4.11 and 4.12, the generated code for the whole expression can be split into four parts:

1. *The computation of the head of the conditional expression.* For this part the trigger thread first has to be split into one thread that starts the computation of the head expression and into another thread that can be used by the body of the conditional expression. In the code sequence denoted by $\mathbf{C}_{E_1}$ the value of the head of the conditional expression is computed. The interface for this part looks as follows:

   - **Input:** $trigger_{E_1}$.
   - **Output:** $result_{E_1}, ly_0$.

   As the result of this expression is used to guide the free variables of the body to the correct side, it has to be immediately split into $n+2$ parallel threads where n is the number of free variables in E_2 and E_3. Therefore, the interface of this splitting in $\mathbf{C}_{split}$ can be described as follows:

   - **Input:** $result_{E_1}$.
   - **Output:** $trigger_i, i = 0, \ldots, n+1$.

2. The next bigger part is the *switching of all free variables* of the body of the conditional expression to the correct branch. For that we assume that the free variables of E_2 and E_3 are $v_1, \ldots, v_n$ with the labels $ly_1, \ldots, ly_n$. The switching of these variables, the trigger and the signal of E_1 is done by $n+2$ guarded SWITCH constructs. In the guards, the free variables have to be synchronized with the splitted result of the expression E_1. Note that the SWITCH instruction has three exits depending on the value in the register that is the first argument of the instruction:

   - The second argument is the exit if this value is TRUE,
   - the third argument is the exit if the value is FALSE,
   - the next line is the exit if the value is ERROR.

   If the evaluation of E_1 returns an error this error value is also the result of the conditional expression (this can be seen from the switching block of the trigger thread, where a COPYI instruction is performed in the error case).

3. The third part of the conditional expression contains the code sequences for the *evaluation of both branches* of the body. Each branch also has an own signal tree, where those variables that are only needed in the other branch and the signal of E_1 are synchronized. The interface of the expressions $\mathbf{C}_{E_j}$ (where $j = true, false$) looks as follows:

   - **Input:** $l_j, l_{j,i}$ for those i where v_i is a free variable of E_j.
   - **Output:** $result_{E_j}, signal_{E_j}$.

   The signal trees $\mathbf{C}_{SignalTree_{E_j}}$ that synchronize the superfluous variables and the signals have the following interface:

trigger :	SPLIT	$trigger_D, trigger_E$	; Evaluate def. and expr. in parallel
	$\mathbf{C}_D$		; Definition Sequence
	$\mathbf{C}_E$		; Expression that yields value of construct
	$\mathbf{C}_{SignalTree}$		; Synchronize all signals

Figure 4.25: ADAM Assembler Code of a Local Definition: $\{D\}E$

- **Input:** $signal_{E_j}, l_{j,i}$ for those i where v_i is not a free variable of E_j.
- **Output:** $signal'_{E_j}$.

The code sequence $\mathbf{C}_{SignalTree_{error}}$ for synchronizing all free variables in the error case has the following interface:

- **Input:** $l_{error,k}, k = 0, \ldots, n$.
- **Output:** *signal*.

4. The last part in the code of Figure 4.24 realizes the *merging of the result and the signal outputs* of both branches into the result and signal output of the conditional expression. Again we can specify the behaviour of this part by describing the input and the output of this merging:

   Merging of the results of both branches:

   - **Input:** $result_{E_{true}}, result_{E_{false}}$.
   - **Output:** *result*.

   Merging of the signals of both branches:

   - **Input:** $signal'_{E_{true}}, signal'_{E_{false}}$.
   - **Output:** *signal*.

   Furthermore, in the merging of the results the actual result value is copied to the correct register r.

Local Definitions

The assembler code of a local definition consists of the code sequence for the definition sequence ($\mathbf{C}_D$) and the code sequence for the expression ($\mathbf{C}_E$). Furthermore, the signals have to be synchronized in the signal tree $\mathbf{C}_{SignalTree}$. Due to the SPLIT instruction in the first line of Figure 4.25 these two parts will be evaluated in parallel. The interface for the different parts looks as follows:

$\mathbf{C}_D$:

- **Input:** $trigger_D$.
- **Output:** $signal_D$.

trigger :	ALLOC	$n, +1$	; Allocate memory for tuple
	RECEIVE	$r, +1$	; Get pointer to memory in r
	SPLIT	$result, split$	; Return pointer and
			; give it to next instructions
	$\mathbf{C}_{split}$		; Splittings for parallel eval.
			; of tuple fields
	$\mathbf{C}_{E_i}$		; $i = 1, \ldots, n$
$result_{E_i}$:	WRITE	$r_i, r, i-1$	; Write value in tuple field
	JUMP	end_i	; Signal output of Write
	$\mathbf{C}_{SignalTree}$		; Synchronize signals and ends

Figure 4.26: ADAM Assembler Code of a Tuple Construction: $[E_1, \ldots, E_n]$

$\mathbf{C}_E$:

- **Input:** $trigger_E$.
- **Output:** $result, signal_E$.

$\mathbf{C}_{SignalTree}$:

- **Input:** $signal_D, signal_E$.
- **Output:** $signal$.

Tuple Construction

Figure 4.26 shows the assembler code that constructs a tuple consisting of n arguments that are evaluated in the code sequences $\mathbf{C}_{E_i}$ (where $i = 1, \ldots, n$). This code bears a strong resemblance to the code for a function call. However, before the controlflow is split into parallel threads, memory must be allocated for the tuple. As this is done via a side-effect a pointer to the allocated memory arrives in the next line and is stored in the register r. As in EVE data structures are non-strict the pointer can be immediately made available to other parts of the program by performing a jump to *result*.

After this first step the controlflow has to be split into n parallel threads, one for each argument of the tuple construction. This splitting is done in the code sequence $\mathbf{C}_{split}$, which has the following interface:

- **Input:** *split*.
- **Output:** $trigger_{E_1}, \ldots, trigger_{E_n}$.

The argument expressions are evaluated in the code sequences $\mathbf{C}_{E_i}$ (where $i = 1, \ldots, n$). These have the following interface:

	$\mathbf{C}_E$		; Body of the tuple
$result_E$:	READ	$i, r_E, +1$	; Select ith component
	RECEIVE	$r, result$	; Receive result in register r

Figure 4.27: ADAM Assembler Code of a Tuple Selection: $\pi_i\ E$

- **Input:** $trigger_{E_i}$.
- **Output:** $result_{E_i}, signal_{E_i}$.

With the WRITE instruction[15] the result of the i-th argument expression is written into the $(i-1)$-st field of the tuple[16]. The signal of this side-effect instruction arrives at the next line and is immediately forwarded to the signal tree, where all unresolved labels are synchronized. The interface for $\mathbf{C}_{SignalTree}$ is therefore:

- **Input:** $signal_{E_i}, end_i$, where $i = 1, \ldots, n$.
- **Output:** $signal$.

Note that contrary to the code in the function call, here is no explicit synchronization before the WRITE instruction necessary. The reason for this is that the tuple is allocated before the triggers of the argument expressions are split. Therefore, it is guaranteed that the memory for the tuple is available when an expression has computed a result.

Tuple Selection

In the code for a tuple selection of Figure 4.27 the only instruction that has to be applied to the expression that is represented by the code sequence $\mathbf{C}_E$ is a READ instruction of the i-th element of the tuple. The result of this operation is received in register r.

The interface of $\mathbf{C}_E$ is:

- **Input:** $trigger$.
- **Output:** $result_E, signal$.

Function Application

The code in Figure 4.28 looks very similar to the code of a function call. As a matter of fact, the only difference to the code in Figure 4.23 is the part of the function invocation. In the function call only first order functions were possible but here the function is in general the result of a curried expression represented by $\mathbf{C}_E$. The structure of such a curried expression will be discussed in the next section. The interface of this expression is:

[15]This WRITE is a WRITEI instruction if an integer value is written to the tuple field or a WRITEP instruction if a pointer is written to the tuple field.

[16]The counting of the fields of a tuple always starts with 0 instead of 1!

	$\mathbf{C}_{split,trigger}$		; Parallel pathes: Eval. fct-expr.
			; and eval. of all args.
	$\mathbf{C}_E$		; Fct.-expr.; yields a curried expr.
$result_E$:	READ	$0, r_E, +1$	; Extract address of fct.
	APPLY	$r_1, +1$	; Allocate activation record in r_{act}
	SUBMIT	$r_E, r_{act}, send$	; Submit curried arguments to fct.
	RECEIVE	$r, result$	; Receive result in register r
	$\mathbf{C}_{split,send}$		; One split for each arg. expr.
			; to allow the sending of args.
	$\mathbf{C}_{E_i}$		; $i = 1, \dots, n$
$result_{E_i}$:	WAIT	$e_i, +2$	; Synchronize result with
$send_{E_i}$:	WAIT	$e_i, +1$	; activation record
	SEND	r_{E_i}, r_{act}, i	; Send arg. to fct.
	JUMP	arg_{E_i}	; Signal output of sending
	$\mathbf{C}_{SignalTree}$		; Synchronize signals and args

Figure 4.28: ADAM Assembler Code of a Function Application: $E[E_1, \dots, E_n]$

- **Input:** $trigger_E$.
- **Output:** $result_E$, $signal_E$.

To get the actual address of the function the first field of the tuple that is the result of $\mathbf{C}_E$ has to be selected. With the APPLY instruction, which gets this address as it is the next instruction after the READ, an activation record is allocated[17]. The pointer to the allocated activation record is stored in r_{act}, as this register appears in the next line. The SUBMIT instruction sends the curried arguments that might be contained in the tuple that is stored in r_E to this function activation. In the next line the final result of the computation is received in the register r.

The interfaces for the two code sequences that perform splittings are:

$\mathbf{C}_{split,trigger}$:

- **Input:** *trigger*.
- **Output:** $trigger_E, trigger_{E_1}, \dots, trigger_{E_n}$.

$\mathbf{C}_{split,send}$:

- **Input:** *send*.
- **Output:** $send_{E_1}, \dots, send_{E_n}$.

[17]This corresponds to the FASTCALL instruction in a function call.

trigger :	ALLOC	$n+1, +1$	; Allocate memory for tuple
	RECEIVE	$r, +1$	; Receive pointer to tuple in r
	SPLIT	$result, split$	; Return pointer and
			; give it to next instructions
	$\mathbf{C}_{split}$		; Splittings for parallel eval.
			; of tuple fields
$trigger_0$:	CONST	$r_0, +2$	; Fct. descriptor in r_0
	$< f, i >$		; Evaluate fct. descriptor
	WRITE	$r_0, r, 0$	; Write fct. descr. in first component
	JUMP	*end*	
	$\mathbf{C}_{v_i}$		; $i = 1, \ldots, n$
$result_i$:	WRITE	$r_i, r, i-1$	; Write curried arg. in
	JUMP	end_i	; ith component
	$\mathbf{C}_{SignalTree}$		; Synchronize all signals and ends

Figure 4.29: ADAM Assembler Code of a Curried Expression: $< f, v_1, \ldots, v_n >$

Curried Expressions

The code sequence of Figure 4.29 has essentially the same structure as the code for a tuple construction that is presented in Figure 4.26. Therefore, here only the differences to this above code sequence are discussed.

- The size of the tuple is $n+1$, since not only the values of the arguments but also the address of the function has to be stored in that tuple.
- After the allocation of the tuple the controlflow is split into $n+1$ parallel threads in $\mathbf{C}_{split}$, one for each field of the tuple. Therefore, $\mathbf{C}_{split}$ now has the following interface:
  - **Input:** *split*.
  - **Output:** $trigger_0, trigger_1, \ldots, trigger_n$.
- The code block that is labelled with $trigger_0$ writes a value that encodes the address of the function and the parameter number of the first of the n fixed arguments into the first field of the tuple. The encoding of this value is done by the assembler when it finds the expression $< f, i >$.
- In a curried expression the parts of the expression must be simple variables rather than general expressions. This is indicated by using $\mathbf{C}_{v_i}$ instead of $\mathbf{C}_{E_i}$. The interface of $\mathbf{C}_{v_i}$ $(i = 1, \ldots, n)$ is:
  - **Input:** $trigger_i$.

	$\mathbf{C}_E$		; Evaluate body expression
$result_E$:	COPYI	$r, r_E, +1$	; Bring result in register r
	JUMP	*result*	; Return result

Figure 4.30: ADAM Assembler Code of a Simple Definition: $v == E$

- **Output:** $result_i, signal_i$.

As in EVE no explicit currying is allowed this construct will normally be used with $n = 0$, thus only containing the encoded value for the function address[18]. The final signal tree ($\mathbf{C}_{SignalTree}$) has the following interface:

- **Input:** $signal_i, end_i$, where $i = 1, \ldots, n$.
- **Output:** *signal*.

Simple Definitions

In the assembler code of a simple definition the result output of the body of the definition can be directly forwarded to the result output of the whole definition. Analogously, the result register of the body of the definition can be taken as the result register of the whole definition.

The reason for using an explicit JUMP statement instead of performing an indirect jump is the fact that indirect jumps may only lead forward but not backward. However, for circular definitions a backward jump is necessary.

One important fact that can't be seen from the assembler code is that the variable that is defined in this simple definition is bound to the register r and the label *result*. This binding is stored in an environment that will be accessed when another expression has to refer to this variable.

Function Definitions

In Figure 4.31 the assembler code of a function definition is shown. In every such function definition the trigger starts the computation of the function by a jump to the first line. The n arguments of the function are received in the next n lines.

The interface of the body expression ($\mathbf{C}_E$) of that function is:

- **Input:** $trigger_E$.
- **Output:** $result_E, signal_E$.

[18] As $n = 0$ the index i of the first fixed argument is meaningless

trigger :	JUMP	$trigger_E$	; Arrival of trigger and args. (below)
	RECEIVE	r_i, l_i	; $i = 1, \ldots, n$
	$\mathbf{C}_E$		; Evaluate body expression
$result_E$:	RETURN	$r_E, +1$	; Return result value to caller
$signal_E$:	WAIT	$e, +1$	; Synchronize with signal of E
	TERMINATE		; Deallocate activation record

Figure 4.31: ADAM Assembler Code of a Function Definition: $f[v_1, \ldots, v_n] == E$

$\mathbf{C}_{split}$	; One trigger for
	; each definition
$\mathbf{C}_{D_i}$	; $i = 1, \ldots, n$
$\mathbf{C}_{SignalTree}$	

Figure 4.32: ADAM Assembler Code of a Definition Sequence: $D_1, \ldots, D_n$

As soon as the result of the body expression has been computed it can be returned to the caller. It has already been noted that from the dataflow point of view a function definition is connected with its caller via dynamic links. Therefore, this construct has no explicit output label. Instead, a TERMINATE instruction is performed after having synchronized all threads in the construct. This instruction causes the activation record of the current function to be discarded.

Definition Sequences

The assembler code for this construct in Figure 4.32 immediately splits the controlflow in n parallel threads. This is done in $\mathbf{C}_{split}$, which has the following interface:

- **Input:** *trigger*.
- **Output:** $trigger_{D_1}, \ldots, trigger_{D_n}$.

Each thread initiates the computation of one definition. The interface of these definitions ($\mathbf{C}_{D_i}$) is:

- **Input:** $trigger_{D_i}$, where $i = 1, \ldots, n$.
- **Output:** $signal_{D_i}$.

All the signals of the different definitions D_i ($i = 1, \ldots, n$) are synchronized in the last signal tree ($\mathbf{C}_{SignalTree}$) that has the following interface:

- **Input:** $signal_{D_i}$, where $i = 1, \ldots, n$.
- **Output:** *signal*.

Program

As by the definition of the EVE language a program is nothing else but a definition sequence, the generated code will look like the code in Figure 4.32. However, a program is a special case of a definition sequence. First of all it contains exactly one constant (`MAIN`). This definition must not be added to the program file that is generated by the compiler. Instead, the generated assembler code of the body of this `MAIN` constant will be put into an own expression file. All the other definitions in a program are function definitions, which don't have an explicit output label. Therefore, there is no signal tree for synchronization necessary.

Chapter 5

Experiments

In this chapter we will present some experiments that we have made with simple EVE programs that have been compiled with the prototype EVE compiler. These experiments include some benchmarks that have been made with standard functional programs on a single transputer and on a network of sixteen transputers. Furthermore, in this chapter the efficiency of the generated ADAM assembler code is discussed. Some possible sources of optimization are discussed in an own section.

5.1 Benchmarks

In order to estimate the quality of the code generated by the EVE compiler, some typical simple functional programs have been implemented in EVE, compiled with the EVE compiler and executed on an ADAM machine with one and with sixteen transputers.

5.1.1 Benchmark Programs in EVE

In this section we discuss the programs that have been used for the benchmarks with the EVE compiler on the ADAM machine. The documented EVE source code of all benchmark programs can be found in Appendix D.

Fibonacci Numbers

In `Fib.eve` a straightforward program for computing a Fibonacci number is given. As this algorithm uses the straightforward definition of Fibonacci numbers it has exponential complexity.

Quicksort

An EVE algorithm for sorting a list is given in the file `QuickSort.eve`. The presented algorithm uses the principle of recursively partitioning the list into a part that is smaller and into a part that is larger than a certain 'pivot' element.

Program	Parameter	1 Processor	16 Processors	Speed Up
Fibonacci	18	40.4 s	5.0 s	8.1
Quick-Sort	[50,49 ... 1]	44.7 s	6.7 s	6.7
N-Queens	6	130.2 s	11.9 s	10.9
Primes	16000	930.3 s	215.7 s	4.3

Table 5.1: Timings of the Benchmark Programs

N-Queens Problem

In `Queens.eve` an algorithm for solving the N-Queens Problem is presented. This algorithm yields all positions of N queens on a chessboard of the size N×N such that no queen checks another queen on the chessboard. As this is one of the most used benchmark programs in functional programming but as it is not as trivial to realize as the two previous programs there exist several versions of this algorithms that differ in some details. However, we have chosen to use a first order program (i.e. it contains no higher order functions) to implement this algorithm. This algorithm also shows the implementation of some basic list operation functions. In the benchmarks this program was executed for a chessboard of the size 6×6.

Prime Number Generator

In `Prime.eve` a program is given, that generates a list of prime numbers up to a given limit `n`. The presented algorithm is a first order algorithm. Contrary to this algorithm, in `CircPrime` two circular prime number generator algorithms are presented. For a discussion of the latter algorithms and their runtime behaviours see Section 5.2. As an upper limit for the list of prime numbers the value 16000 has been chosen in the benchmarks.

5.1.2 Timings

In this section we present the timings that have been achieved by executing the above programs on an ADAM machine with only one processor ('sequential case') and on an ADAM machine with sixteen processors ('parallel case'). The runtimes for the sequential and for the parallel case are given in Table 5.1. The second column in this table denotes the value of the parameter with which the benchmark program was called. Furthermore, the last column denotes the factor by which the parallel case was faster than the sequential case. All timings are given in seconds.

The results in Table 5.1 show that with a problem that contains much inherent parallelism a speed-up of almost 11 on a network of sixteen processors can be achieved (N-Queens). Although, these speed ups might be not as impressing as speed ups that could be achieved with explicitly parallel programs, it has to be emphasized that these results were achieved by only using the implicit parallelism of the functional program without giving the compiler or the abstract machine additional help in generating code or executing the program[1].

[1] Giving the programmer the additional possibility of submitting such information to the compiler would be one method to potentially increase the efficiency of the generated parallel programs.

Program	1 Processor	16 Processors	Speed Up
Wentworth's algorithm	2.3 s	1.1 s	2.1
Schreiner's algorithm	1.2 s	0.4 s	3.0

Table 5.2: Timings of Two Circular Prime Number Generators

Moreover, the same object code that is generated by the EVE compiler can be used for a sequential and for a parallel execution by just using abstract machines with a different number of processors.

When one compares the absolute times of these tests with the runtimes achieved by hand-written ADAM assembler programs one notices that the latter programs require about 50 to 70 % of the time needed for the execution of a compiled EVE program. For example a hand-written version of the Fibonacci number algorithm with parameter 18 takes 27.9 seconds in the sequential case. That are about 70 % of the time needed by the compiled EVE program as it can seen from Table 5.1. The main reason for this quite low efficiency is the fact that the prototype EVE compiler does not perform any optimizations.

5.2 Comparison of Two Circular Prime Number Generators

In Appendix D in `CircPrime.eve` two versions of a circular prime number generator are presented[2]. One version, which will be called Wentworth's algorithm from now on, is taken from [Wentworth, 1989a] and the other algorithm, which will be called Schreiner's algorithm, is taken from [Schreiner, 1990]. Generally, such a circular prime number generator is more efficient than the straightforward algorithm in `Prime.eve` as the circular algorithm can access previous elements of the list that is being constructed. Hence, for the test whether a number n is prime it is divided by the prime numbers that are smaller than or equal to $\sqrt{n}$. These prime numbers are taken from the list that is the result of the whole computation and that is therefore just being constructed. This is of course more efficient than dividing a number n by all odd numbers smaller than $\sqrt{n}$ as it is done by the function `isPrime` in `Prime.eve`.

Both of the presented algorithms are based on this principle. However, the implementation shows a subtle difference in the parallel runtime behaviour of the programs that has strong influence on the speedup that can be achieved with these algorithms. Table 5.2 shows the execution times of both algorithms in the sequential and in the parallel case as well as the speed up, which is very small in this case as only a list of all prime numbers up to 70 has been computed.

To understand this difference we have to take a closer look at the data dependencies inside both algorithms, which determine their runtime behaviours. At first glance both algorithms have the same global structure as they both create a bidirectional pipeline of 'processes'. In the case of Wentworth's algorithm these processes are those calls to the function `oddPrimesFromTo` for which `n<=limit` holds. The first call of this function with `n>limit`

[2]For a discussion of the notion of 'circularity' see Section 2.1.

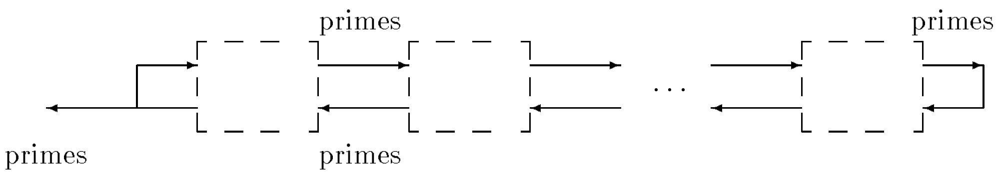


Figure 5.1: Global Structure of Both Prime Number Generators

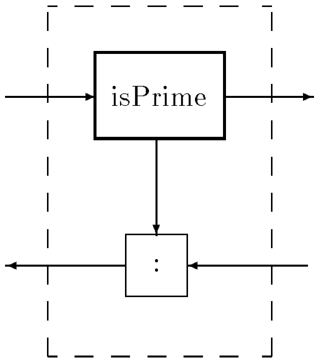


Figure 5.2: Structure of a Process in Wentworth's Prime Number Generator

builds the right end of the pipeline where the list of primes that has been built is pushed back. In the case of Schreiner's algorithm these processes consist of one call to `sieve` and one call to `cons`. Again the condition `n>limit` determines the end of the pipeline. This global structure of both algorithms can be seen in Figure 5.1.

However, when we examine the interior of the processes in the two algorithms we can see the significant difference between them. Figure 5.2 shows the structure of a process in Wentworth's algorithm where `n<=limit`. The characteristic feature of such a process is the fact that when the input arrives from the left the function `isPrime` has to be computed before the input can be forwarded to the next process, which is nothing but the recursive call inside `oddPrimesFromTo`. However, the computation of `isPrime` is the most time consuming part of the whole function, which is denoted in the figure by the thick box around `isPrime`. Hence, the whole computation is sequentialized by the data dependency between the head of the if-expression and the branches in the if-expression, that forms the body of the function `oddPrimesFromTo`.

Contrary to this, in Schreiner's algorithm one process consists essentially of function calls to `sieve` and to `cons` (again `n<=limit` in these function calls). The structure of such a process can be seen in Figure 5.3. The crucial difference to the process in Figure 5.2 is the fact that `cons` is the time consuming function that has to test whether the actual `n` is prime or not. When we examine the definition of `sieve` we see that there immediately after checking the termination condition of the pipeline a recursive call can be performed. Therefore, the

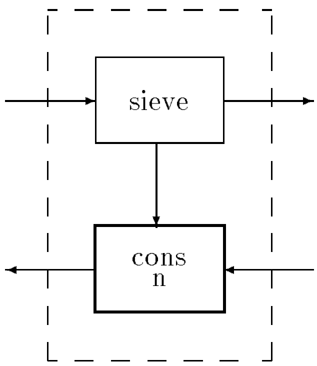


Figure 5.3: Structure of a Process in Schreiner's Prime Number Generator

input can be immediately forwarded to the next process in the pipeline, thereby creating a large number of parallel processes. On the way back from the right end of the pipeline in the `cons` function the time consuming computations are performed in parallel in all created processes. In the source code of the program this behaviour can be seen as the test for primality is put from the head of the if-expression (in the case of Wentworth's algorithm) into the else-branch of the algorithm where it does not block the recursive call of `sieve`[3].

From this case study one can see that it would be naive to believe that with a parallelizing compiler like the one presented in this thesis one can automatically get efficient parallel programs. In this approach of automatic parallelization the parallelism is restricted by the data dependencies in a program. Therefore, one can say that with functional languages the problem of parallelization is reduced to the problem of eliminating or weakening the data dependencies in a program. However, it should be noted that the notion of data dependency is still a notion of the functional specification of the program. Hence, the elimination of data dependencies can be treated in a similar way as other transformations on the functional program without leaving this model of computation. As it is far easier to apply mathematical transformations to functional programs than to imperative programs, one has a better chance to increase the parallel efficiency of functional programs by performing mathematical transformations on these programs within an optimizing compiler (e.g. by using some transformations that weaken strong data dependencies). Summarizing we can say that *the data dependencies in a functional program build the interface to the operational behaviour of the program with respect to parallelism.*

5.3 Sources of Optimization

It has already been mentioned that the prototype EVE compiler does not perform any optimizations on the generated ADAM assembler code. However, by looking at the generated code some inefficiencies become obvious. In this section we will discuss some ideas for performing optimizations on the generated code. However, there are many optimization techniques for sequential languages that could be performed on this code. But we are here

[3]Remember that the definitions in a local definition are evaluated in parallel with the body of the construct.

only interested in those optimizations that directly affect the parallel behaviour of the program. As our approach is based on the dataflow model this mainly means that we will try to group blocks of code together to get bigger sequential parts that can then be executed in parallel with each other. This is essentially the same idea that has also lead to the research in the area of 'macro-dataflow' (see [Sarkar and Hennessy, 1986]).

To explain the major inefficiencies in the code produced by the EVE compiler we show in Figure 5.4 the ADAM assembler code that is generated by the EVE compiler for the expression

```
{ a : INT == 1 ;
  double[x] : INT[INT] == 2*x }
a + double[5];
```

Note that the function `double` has been lifted out of the local definition and put on top level. Figure 5.5 shows the code that is generated for this function.

From this code several inefficiencies become obvious. The most obvious inefficiency in the code is the large number of SPLIT and WAIT instructions in the code. Sometimes one of the threads of control that is created with a SPLIT instruction is immediately synchronized without performing some useful computation inbetween. The reason for these 'empty' threads in which no computation is performed is the translation rule for simple variables. As it can be seen from the dataflow graph that corresponds to a simple variable (see Figure 4.7 in Section 4.4.1) no operation has to be performed in the two independent threads. However, in each construct the parallel threads have to be synchronized in such a way that the construct has exactly two output arcs (from the dataflow point of view). For example, whenever a binary operation is applied on two variables, two parallel threads are created only to see that in these parallel threads no computation has to be performed. Hence, the parallel threads are immediately synchronized with each other. As this behaviour essentially holds for every appearance of a variable a large number of superfluous SPLIT and WAIT instructions is introduced. The major task of an optimization must be to eliminate these superfluous synchronization statements. This can be achieved in several different ways:

- As all arithmetic/boolean operations are strict in all of their arguments, it is safe to evaluate these arguments before the operation itself is performed. From the dataflow point of view this means that the trigger input can be directly forwarded to the signal output and the free variables can be directly fed to the operations. From the controlflow point of view that means that there is only one thread of control necessary. Figure 5.6 shows a dataflow graph as it would be derived from the EVE expression `x*(y+z)` via the rules in Section 4.4.1. When one examines the splitting of the trigger arcs it becomes obvious that this trigger is entirely superfluous here, as after each splitting it is immediately synchronized in a signal tree. Therefore, one can optimize this dataflow graph yielding the graph that is shown in Figure 5.7. In this graph the trigger is directly forwarded to the signal output.

- The above transformation can eliminate many superfluous synchronization statements but it is restricted to purely arithmetic/boolean expressions. As soon as a user defined function appears in an expression this step can not be used in this straightforward way any more.

```
       SPLIT     l3,   l4
;
; New Definition -----------
;
l3:    CONST     4,    2
       1
       SPLIT     l7,   l8
;
l7:    COPYI     3,    4,     1
       JUMP      l6
;
; End of Definition -------------
;
; Binary %%%%%%%%%%%%
;
l4:    SPLIT     l9,   l10
;
; CALL %%%%%%%%%%%%%
;
l10:   SPLIT     l17,  l18
l18:   FASTCALL  double,      1
       COPYI     6,    6,     l14
       COPYI     7,    7,     l15
;
l17:   CONST     8,    2
       5
       SPLIT     l19,  l20
;
l19:   WAIT      4,    2
l14:   WAIT      4,    1
       SEND      8,    6,     1
       JUMP      l13
l13:   WAIT      5,    l16
l20:   WAIT      5,    l16
;
; end of CALL $$$$$$$$$$$$$
;
l6:    WAIT      1,    2
l15:   WAIT      1,    1
       COPYI     5,    3,     1
       ADD       5,    7,     l11
l9:    WAIT      2,    l12
l16:   WAIT      2,    l12
;
; end of Binary $$$$$$$$$$$$$
;
l11:   COPYI     1,    5,     1
       RETURN    1,    1
l8:    WAIT      7,    1
l12:   WAIT      6,    1
```

Figure 5.4: ADAM Assembler Code for a Local Definition in EVE

```
FUNCTION double(5,5)

       JUMP      l6
       COPYI     2,    2,     l8
;
; Binary %%%%%%%%%%%%
;
;
l6:    SPLIT     l9,   l10
;
l9:    CONST     4,    2
       2
       SPLIT     l13,  l14
;
l13:   WAIT      1,    2
l8:    WAIT      1,    1
       COPYI     3,    4,     1
       MUL       3,    2,     l11
l14:   WAIT      2,    l12
l10:   WAIT      2,    l12
;
; end of Binary $$$$$$$$$$$$$
;
l11:   RETURN    3,    1
l12:   WAIT      4,    1
       TERMINATE
END double
```

Figure 5.5: ADAM Assembler Code for an EVE Function

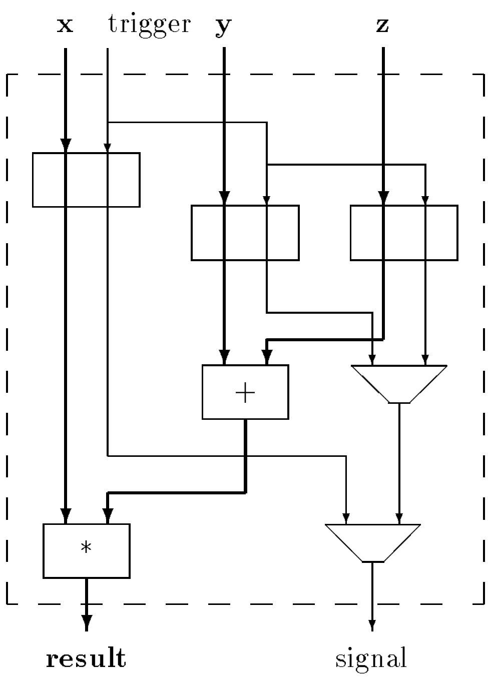


Figure 5.6: Dataflow Graph of `x*(y+z)`

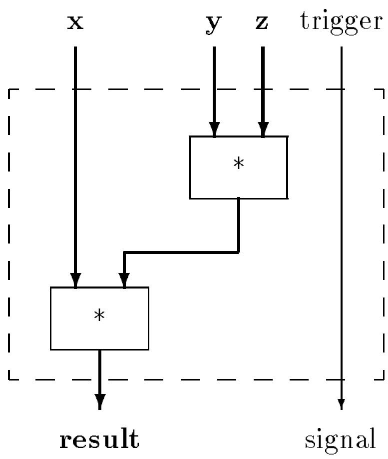


Figure 5.7: Optimized Dataflow Graph of `x*(y+z)`

However, one can perform a *strictness analysis* on all user defined functions. This strictness analysis yields the information in which parameters the function is strict. This information can then be used in a function call to evaluate the strict arguments of the function sequentially before the function itself is started. Thereby, the parallelism in a program is reduced and the grain size of the sequential blocks is increased. For a discussion of the notion of strictness analysis and its applicability for the compilation of functional programming languages see [Burn, 1991].

- Another way to eliminate 'empty' threads in the generated code would be to use a *peephole optimizer*. The idea with this optimizer is to scan the generated code with a window and to search in this window for patterns of instructions that are known to be inefficient. These patterns can then be replaced by more efficient instructions. The advantage of such an optimizer is that it is quite simple to implement. The major disadvantage is obviously that only local optimizations can be performed. However, as it is shown in e.g. [Augustsson, 1987] even such a simple optimizer can increase the efficiency of a program significantly.

  If one wants to use such a peephole optimizer for the optimization of parallel code one has to extend its concept a little bit. In the sequential case it suffices to use only one window as one has the additional information that there is only one thread of control in the program. However, as there are several threads of control in a parallel program there will be also several windows necessary to perform sensible optimizations. Especially, for eliminating 'empty' threads in a program such an optimizer can be very useful as for this elimination only local information about the program is necessary.

- If one wants to build bigger, sequential blocks in a program one could try to combine expressions like the branches of a conditional expression into sequential blocks. However, in order to get blocks of an appropriate size one must be able to estimate the execution time that is needed for every statement. The compile time analysis that performs such an estimation is called *weight analysis*. This process helps to build sequential blocks of an optimal size to get an efficient program. However, as it is especially difficult to analyze the costs for executing recursive functions, the value of the results of this analysis is quite limited.

Chapter 6

Conclusion

The main goal of the ADAM & EVE project, of which this thesis is a part, was to study the behaviour of functional languages that have been compiled to the assembler code of an abstract dataflow machine ADAM. As this thesis shows, dataflow graphs can be used to get controlflow programs that make the implicit parallelism of a functional language explicit. This transformation into explicit parallelism that is expressed in a language similar to that of a von-Neumann processor is an important step to achieve an efficient parallelization. The main reasons why the resulting parallel programs that have been presented in Chapter 5 are nevertheless rather inefficient are that

- the prototype EVE compiler does not perform any optimizations and that
- the resulting ADAM code has to be interpreted.

However, as it has been described in Section 5.3 there are many promising possibilities to improve the efficiency of the generated ADAM assembler code. Furthermore, functional programming languages offer the possibility to perform transformations on the intermediate code in a quite straightforward way. In the EVE compiler this is only used in the λ-lifting transformation. However, there could also be some transformations used that directly affect the parallel behaviour of the generated program[1]. The second inefficiency, namely the usage of an interpreter as an implementation of the abstract machine, has been accepted as the main goal of the project was not efficiency but to show the principle applicability of the dataflow model as an abstract machine. However, there is currently some work done at RISC that aims at improving the efficiency of this approach by using a library of C functions instead of an abstract machine to simulate dataflow behaviour (see [Schreiner, 1991]). Thus one could say that here a development from interpreted to *compiled dataflow code* takes place.

As another important outcome of this project it was described how the dataflow model can be used to understand the parallel behaviour of functional programs. Section 5.2 shows that the dataflow model can visualize the data dependencies in a functional program and thereby also the parallel behaviour of the program. Therefore, the dataflow model is not only a good model for the parallel implementation of functional programming languages, but it is also

[1]Note that the λ-lifting transformation is mainly intended for sequential implementations of functional languages and does not improve the parallel behaviour of the generated program.

a good model for understanding the operational behaviour of functional programs. This is of course very important for the programmer who has to write parallel functional programs as he can with this knowledge "parallelize" his functional program by weakening the data dependencies in the program.

The main insight from the compiler design point of view is the fact that dataflow graphs form a useful intermediate language in the parallelization of functional languages. Although it is very intuitive to use this intermediate language when the source language of the compilation process is a functional language, this approach need not be restricted only to functional languages.

Summarizing, it can be said that the dataflow model proves useful in almost all steps of the parallel implementation of functional programs.

Appendix A

Attribute Grammar of the First Pass

This appendix gives a formal description of the first pass in the compilation process. For this formal description an *attribute grammar* is used. This way of describing a compilation process was introduced by D. E. Knuth in [Knuth, 1968]. One advantage of this approach is that an imperative program can be quite easily derived from an attribute grammar.

The main idea of an attribute grammar is to extend the given grammar of the language by semantic actions that are executed interleavingly with the procedures for parsing non-terminal symbols of the grammar. In order to establish a communication between the non-terminal symbols and the semantic procedures *attributes* are added to these non-terminal symbols. With this communication it is possible to build up tables and other data structures that are needed by the next passes in the compilation. The actual output of the first pass consists of

- an **abstract syntax tree** that represents the source program,
- a **symbol table** that contains all objects of the source program,
- a collection of **types**.

The whole formal description of the first pass consists of four parts:

1. The **context-free syntax** of EVE, with which a parser that is based on the principle of the recursive descent can be constructed. Hence, the presented grammar has no left recursion and no alternatives with the same terminal starting symbols. This grammar is obtained from the grammar in 3.2.17 by performing a left factorization on the alternatives of *Type* that start with the terminal symbol *Ident*. To perform this left factorization one has to introduce the new non-terminal symbols *SimpleOrFctType* and *TCFDefinition*. Furthermore, one has to change the definitions of the non-terminal symbols *FunctionType*, *Type*, *TypeDefinition*, *ConstantDefinition*, *FunctionDefinition* and *Definition*.
2. The **declaration of the objects** that are used in the attribute grammar. This includes the type declarations of all important data structures except those structures

that are implemented as abstract data types [1]. Especially, the declarations of all the attributes that are used in the attribute grammar are given here.

3. The description of the **semantic procedures** that are used in the attribute grammar. Again, the semantic procedures that operate on abstract data types are not described here. To get a description about them see Appendix E.

4. The **attribute grammar**. This part can already be read like an imperative program. Note that there are no 'meta-keywords' **sem** and **endsem** as they are often used to separate semantic actions from the actual grammar. Instead, syntactic and semantic parts are distinguished by indentation. All parts referring to the syntax of EVE start in the first column. All semantic actions and context assertions are indented.

   In the attribute grammar at some points the meta-keyword **CC** with a succeeding boolean expression can be found. This marks a context condition that must be true at that point. If it is not true, the compiler will yield a semantic error.

A.1 Altered Context-Free Syntax of EVE

Types

IdentList	=	*Ident* { , *Ident* } .
SimpleType	=	*Ident* .
TypeList	=	*SimpleType* { , *SimpleType* } .
Field	=	*Ident* : *SimpleType* .
FieldList	=	*Field* { , *Field* } .
Variant	=	*Ident* `==` *SimpleType* .
VariantList	=	*Variant* { , *Variant* } .
ListType	=	`LIST OF` *SimpleType* .
TupleType	=	`TUPLE OF` *FieldList* .
UnionType	=	`UNION OF` *VariantList* .
FunctionType	=	`[` *TypeList* `]` .
SimpleOrFctType	=	*SimpleType* [*FunctionType*] .
Type	=	*SimpleOrFctType* | *ListType* | *TupleType* | *UnionType* .

Definition Statements

TypeDefinition	=	[`==` *Type*] .
ConstantDefinition	=	: *SimpleType* [`==` *Expression*] .
FunctionDefinition	=	`[` *IdentList* `]` : *SimpleOrFctType* [`==` *Expression*] .
TCFDefinition	=	*TypeDefinition* | *ConstantDefinition* | *FunctionDefinition* .
Definition	=	[*Ident* *TCFDefinition*] .
DefinitionSequence	=	*Definition* { ; *Definition* } .

[1] For a specification of these abstract data types and of the procedures that operate on them see Appendix E.

Expressions

Expression = *ListExpr* | `{` *DefinitionSequence* `}` *Expression* | `IF` *Expression* `THEN` *Expression* `ELSE` *Expression* | `CASE` *Expression* `{` *Field* `}` *Expression* { `{` *Field* `}` *Expression* } [`ELSE` *Expression*].
ListExpr = *CompoundExpr* { `::` *Factor* } .
CompoundExpr = *SimpleExpr* [*Relation* *SimpleExpr*] .
Relation = `=` | `#` | `<` | `<=` | `>` | `>=` | `<>` .
SimpleExpr = [`+` | `-`] *Term* { *AddOp* *Term* } .
AddOp = `+` | `-` | `|` .
Term = *Factor* { *MulOp* *Factor* } .
MulOp = `*` | `/` | `%` | `&` .
Factor = *Number* | *Character* | *String* | *Designator* { *ArgumentList* | *Selector* } | `(` *Expression* `)` | `~` *Factor* .
Designator = *Ident* [`$` *Ident*] .
ArgumentList = `[` [*Expression* { `,` *Expression* }] `]` .
Selector = `.` *Ident* .

Programs

Program = *DefinitionSequence* .

A.2 Declarations of Objects

A.2.1 Constants

maxIdentListLen	=	16
maxSTypeListLen	=	16
maxIdTypeListLen	=	16
maxNameTypeListLen	=	16
maxTypeListLen	=	16
maxIdASTypeListLen	=	16
maxNumIdASTypeListLen	=	16
maxFieldListLen	=	16
maxVariantListLen	=	16
maxASTypeListLen	=	16
maxFlagListLen	=	16

A.2.2 Types

```
IType = RECORD                          — ident with corresponding
          name, type : CARDINAL;          name of type
        END;

Ntype = RECORD                          — ident with corresponding type
```

```
          name : CARDINAL;
          type : EType;
        END;

IdASType = RECORD
             name : CARDINAL;
             ast : ASType;
           END;

NumIdASType = RECORD                       — for the bodies of the branches
                number, name : CARDINAL;   of a CASE Expression
                ast : ASType;
              END;

OptionalASType = RECORD                    — AST of the optional ELSE
                   CASE exist : BOOLEAN OF branch in a CASE Expression
                     TRUE : ast : ASType;
                   END;
                 END;

FwType = RECORD                            — the type of a forward definition
           CASE isForward : BOOLEAN OF
             TRUE : type : EType;
           END;
         END;

Flags = (pointer, int, none);

RelationType = (eqRelTy, ltRelTy, leRelTy, gtRelTy, geRelTy, neRelTy);

ArithmType = (minusArTy, plusArTy, orArTy, timesArTy, divArTy, modArTy, andArTy);

AtomType = (charAtom, idAtom, numberAtom);

IdList          = ARRAY [1..maxIdentListLen] OF CARDINAL;
STypeList       = ARRAY [1..maxSTypeListLen] OF CARDINAL;
IdTypeList      = ARRAY [1..maxIdTypeListLen] OF IType;
NameTypeList    = ARRAY [1..maxNameTypeListLen] OF NType;
ETypeList       = ARRAY [1..maxTypeListLen] OF EType;
IdASTypeList    = ARRAY [1..maxIdASTypeListLen] OF IdASType;
NumIdASTypeList = ARRAY [1..maxNumIdASTypeListLen] OF NumIdASType;
ASTypeList      = ARRAY [1..maxASTypeListLen] OF ASType;
FlagList        = ARRAY [1..maxFlagListLen] OF Flags;
```

A.2.3 Attributes

name, nameTy, len, len1, nr	:	CARDINAL;
idList	:	IdList;
sTypeList	:	STypeList;
idType	:	IType;
idTypeList	:	IdTypeList;
eType, eTypeExpr, eTypeIn, eTypeOut, eType0, eType1, eType2, eTypeU, eTypeHead, eTypeThen, eTypeElse	:	EType;
typeList	:	ETypeList;
x, ast, ast0, ast1, ast2, astHead, astThen, astElse	:	ASType;
fw	:	FwType;
rel	:	RelationType;
add, mul	:	ArithmType;
c	:	CHAR;
isUnionFct, isHigherOrder	:	BOOLEAN;

A.2.4 Further Variables

name1, i, j, lenT	:	CARDINAL;
eTypeRet, listType, eTypeVar	:	EType;
defTypeList	:	ETypeList;
nameType	:	NType;
nameTypeList	:	NameTypeList;
x0, nrAST, astLambda	:	ASType;
def, cells, arg, asts	:	ASTypeList;
elseBody	:	OptionalASType;
body	:	IdASTypeList;
obj	:	Object;
op	:	Operand;
flagList	:	FlagList;
fwd, renamed, sign, loopEntered	:	BOOLEAN

A.3 Semantic Procedures

Join (↓ idList,↓ typeList,↓ len,↑ nameTypeList);

— Combine two lists of equal length to a list of pairs (the `zip` function of usual functional languages).

```
BEGIN
  FOR i:=1 TO len DO
    nameTypeList[i].name:=idList[i];
    nameTypeList[i].type:=typeList[i];
  END;
END Join;
```

StringToList (↓ saddr, ↓ slen, ↑ x);

— Convert a string into the abstract syntax tree that represents this string.

```
VAR str : LongString;
    ast : ASTypeList;

BEGIN
  FetchString (↓ saddr,↓ slen,↑ str);
  FOR i := 0 TO slen–1 DO
     MakeASTAtom (↓ charAtom,↓ str[i],↓ noCardVal,↑ ast[i + 1]);
  END;
  MakeASTList (↓ ast,↓ slen,↑ x);
END StringToList;
```

forwarded (↓ obj,↓ op) ↑ BOOLEAN

— Test whether there exists a forward definition for the given object on the same level.

```
BEGIN
  RETURN (obj<>noObject AND op.level=CurrentLevel())
END forwarded;
```

CreateDummyIdList (↓ len,↑ idList)

— Create a list of empty identifieres of length `len` (this is needed for forward definitions of functions).

```
BEGIN
  FOR i := 1 TO len DO
     idList[i] := dummyName;
  END;
END CreateDummyIdList;
```

CheckTypes (↓ idList,↓ typeList,↓ len,↓ name,↓ eType,↓ eTypeRet);

— Check whether the names and types given in the lists `idList` and `typeList` (`name` and `eType` are the name and the type of of the return value) are the same as the names and types that are derived from the function type `eTypeRet`.

```
VAR idlistOk, typeListOk : BOOLEAN;
    nameRet              : CARDINAL;
    eTypeRetOrig         : EType;
    typeList1            : ETypeList;
    idList1              : IdList;

BEGIN
  GetParTypeList(↓ eType,↑ idList1,↑ typeList1,↑ len1)
  GetReturnType(↓ eType,↑ eTypeRetOrig)
```

```
  GetReturnName(↓ eType,↑ nameRet)
  TestIdList(↓ idList,↓ len,↓ idList1,↓ len1,↑ idListOk)
  TestTypes(↓ typeList,↓ len,↓ typeList1,↓ len1,↑ typeListOk)
  CC: (idListOk AND typeListOk AND (name=nameRet) AND
        equalType(eTypeRet,eTypeRetOrig))
END CheckTypes;
```

flag (↓ eType) ↑ Flags

— Determine whether the type `eType` denotes a compound data structure, which is represented by a pointer in the ADAM machine, or whether it denotes an atomic data structure, which is represented as an integer value.

```
BEGIN
  IF  isListType(eType) OR isTupleType(eType) OR isFctType(eType)
      THEN RETURN pointer
      ELSE RETURN int
  END;
END flag;
```

TypesToFlags (↓ typeList,↓ len,↑ flagList)

— Apply the `flag` function on the list `typeList` yielding `flagList`.

```
BEGIN
  FOR i := 1 TO len DO
     flagList[i] := flag(typeList[i]);
  END;
END TypesToFlags;
```

A.4 Attribute Grammar of the First Pass

Types

$IdentList_{\uparrow\text{idList}\uparrow\text{len}}$ =

len := 0

$Ident_{\uparrow\text{name}}$

INC(len);
idList[len] := name

{ ,

CC: len < maxIdentListLen

$Ident_{\uparrow\text{name}}$

INC(len);
idList[len] := name

}

$SimpleType_{\uparrow\text{name}}$ =

$Ident_{\uparrow \mathrm{name}}$

$TypeList_{\uparrow \mathrm{sTypeList} \uparrow \mathrm{len}}$ =
    len := 0
$SimpleType_{\uparrow \mathrm{name}}$
    INC(len);
    sTypeList[len] := name
{ ,
    CC: len < maxSTypeListLen
$SimpleType_{\uparrow \mathrm{name}}$
    INC(len);
    sTypeList[len] := name
}

$Field_{\uparrow \mathrm{idType}}$ =
Ident$_{\uparrow \mathrm{name}}$
    idType.name := name
    DisableRenaming
: $SimpleType_{\uparrow \mathrm{name}}$
    EnableRenaming
    idType.sType := name

$FieldList_{\uparrow \mathrm{idTypeList} \uparrow \mathrm{len}}$ =
    len := 0
$Field_{\uparrow \mathrm{idType}}$
    INC(len);
    idTypeList[len] := idType;
{ ,
    CC: len < maxFieldListLen
$Field_{\uparrow \mathrm{idType}}$
    INC(len);
    idTypeList[len] := idType
}

$Variant_{\uparrow \mathrm{idType}}$ =
Ident$_{\uparrow \mathrm{name}}$
    idType.name := name
== $SimpleType_{\uparrow \mathrm{name}}$
    idType.type := name

$VariantList_{\uparrow \mathrm{idTypeList} \uparrow \mathrm{len}}$ =
    len := 0
$Variant_{\uparrow \mathrm{idType}}$

INC(len);
idTypeList[len] := idType;

{ ,

CC: len < maxVariantListLen

Variant$_{\uparrow\mathrm{idType}}$

INC(len);
idTypeList[len] := idType

}

ListType$_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$ =
LIST OF SimpleType$_{\uparrow\mathrm{name}}$

GetType(↓ name,↑ eType1)
MakeListType(↑ eType,↓ eType1,↓ fw)

TupleType$_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$ =
TUPLE OF *FieldList*$_{\uparrow\mathrm{idTypeList}\uparrow\mathrm{len}}$

FOR i := 1 TO len DO
  SearchType(↓ idTypeList[i].type, ↑ nameTypeList[i].type)
  nameTypeList[i].name := idTypeList[i].name
END
MakeTupleType(↑ eType, ↓ nameTypeList, ↓ len, ↓ fw)

UnionType$_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$ =
UNION OF *VariantList*$_{\uparrow\mathrm{idTypeList}\uparrow\mathrm{len}}$

FOR i := 1 TO len DO
  SearchType(↓ idTypeList[i].type,↑ nameTypeList[i].type)
  nameTypeList[i].name := idTypeList[i].name
END
MakeUnionType(↑ eType, ↓ nameTypeList,↓ len, ↓ fw)

SimpleOrFunctionType$_{\uparrow\mathrm{eType}\uparrow\mathrm{typeList}\uparrow\mathrm{len}}$ =
SimpleType$_{\uparrow\mathrm{name}}$

FindObject(↓ name,↑ obj,↑ op)
CC: obj<>noObject
len := 0
eType := op.type

(ε

IF op.kind=fctKind THEN
  CC: isFct(op.type)
  ExtractParTypes(↓ name,↑ typeList,↑ len)
END

| *FunctionType*$_{\uparrow\mathrm{typeList}\uparrow\mathrm{len}}$)

$\mathit{FunctionType}_{\uparrow\mathrm{typeList}\uparrow\mathrm{len}}$ =
[$\mathrm{TypeList}_{\uparrow\mathrm{sTypeList}\uparrow\mathrm{len}}$
FOR i := 1 TO len DO
SearchType(↓ sTypeList[i],↑ typeList[i])
END
]

$\mathit{Type}_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$ =
$\mathit{SimpleOrFctType}_{\uparrow\mathrm{eType}\uparrow\mathrm{typeList}\uparrow\mathrm{len}}$
IF len>0 THEN
eTypeRet := eType;
CreateDummyIdList(↓ len,↑ idList)
Join(↓ idList,↓ typeList,↓ len,↑ nameTypeList)
MakeFctType(↑ eType,↓ eTypeRet,↓ dummyName,
↓ nameTypeList,↓ len,↓ fw)
END
| $\mathit{ListType}_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$
| $\mathit{TupleType}_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$
| $\mathit{UnionType}_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$

Definition Statements

$\mathit{TypeDefinition}_{\downarrow\mathrm{name}\uparrow\mathrm{eType}}$ =
FindObject(↓ name,↑ obj,↑ op)
CC: obj=noObject OR
(obj<>noObject AND op.level<CurrentLevel()) OR
(obj<>noObject AND op.forward AND op.kind=typeKind
AND equalType(op.Type,anyType));
IF existInSymbolTable(name) THEN
Rename(↓ name,↓ CurrentLevel())
END — if necessary rename new type
fw.isForward := forwarded(↓ obj,↓ op)
IF fw.isForward THEN
fw.type := op.type
ELSE
NewType(↓ name)
END — name in symbollist (currLevel)
(== $\mathit{Type}_{\downarrow\mathrm{fw}\uparrow\mathrm{eType}}$
IF fw.isForward THEN
ResolveForwardType(↓ name)
ELSE
AssignType(↓ name,↓ eType)
END
| ε
CreateType(↑ eType)
MakeForwardType(↓ name,↓ eType)

```
)


ConstantDefinition↓name↑eType↑x =
        x := emptyAST
        IF existInSymbolTable(↓ name) THEN
            Rename(↓ name,↓ CurrentLevel())
        END                                        — if necessary rename new const.
: SimpleType↑nameType
        SearchType(↓ nameType,↑ eType)
        FindObject(↓ name,↑ obj,↑ op)
        IF name=nameOfMain THEN
            obj := objOfMain
            op := opOfMain
            AssignType (↓ name,↓ eType)
        END
        CC: obj=noObject OR
             (obj<>noObject AND op.level<CurrentLevel()) OR
             (obj<>noObject AND op.forward AND op.kind=constKind
             AND equalType(op.Type,eType));
        fwd := forwarded(↓ obj,↓ op)
        IF NOT fwd THEN
            NewConst(↓ name,↓ eType)
        END                                        — name in symbollist (currLevel)
( == Expression↑ast1↑eType1
        CC: equalType(↓ eType,↓ eType1)
        MakeASTDef(↓ name,↓ ast1,↓ flag(eType),↑ x)
        IF fwd THEN
            ResolveForwardConstant(↓ name,↓ x)
        ELSE
            AssignAST(↓ name,↓ x)
        END
|ε
        MakeForwardConstant(↓ name)
)


FunctionDefinition↓name↑eType↑x =
        x := emptyAST
        FindObject(↓ name,↑ obj,↑ op)
        fwd := forwarded(↓ obj,↓ op)
        CC: obj=noObject OR
             (obj<>noObject AND op.level<CurrentLevel()) OR
             (obj<>noObject AND op.forward AND op.kind=fctKind)
        IF name=nameOfMain THEN
            obj := objOfMain
            op := opOfMain
        END
```

```
        IF existInSymbolTable(name) AND name<>nameOfMain THEN
            Rename(↓ name,↓ CurrentLevel())
        END                                   — if necessary rename new const.
[ IdentList↑idList↑len ] :
        CC: len>0
SimpleOrFctType↑eType↑typeList↑len1
        IF len1>0 THEN                        — Explicit FctType
            CC: len=len1                      — eType is return type
            eTypeRet := eType
            Join (↓ idList,↓ typeList,↓ len,↑ nameTypeList);
            IF NOT fwd THEN                   — i.e. new fct
                MakeFctType(↑ eType,↓ eTypeRet,↓ name,
                            ↓ nameTypeList,↓ len,↓ noForward)
                NewFct(↓ name,↓ eType)
            ELSE                              — i.e. prev. forward
                IF name=nameOfMain THEN
                    MakeFctType(↑ eType,↓ eTypeRet,↓ name,
                                ↓ nameTypeList,↓ len,↓ noForward)
                    SetMAINFctType (↓ eType, ↓ objOfMain, ↓ opOfMain)
                ELSE
                    eType := op.Type
                    CheckTypes (↓ idList,↓ typeList,↓ len,
                                ↓ name,↓ eType,↓ eTypeRet)
                END
            END
        ELSE                                  — eType=type of fct
            SetParNames(↓ name,↓ idList,↓ len,↓ eType)
            GetParTypeList(↓ eType, ↑ idList,↑ typeList,↑ lenT)
            GetReturnType(↓ eType,↑ eTypeRet)
            Join (↓ idList,↓ typeList,↓ len,↑ nameTypeList)
            IF NOT fwd THEN                   — i.e. new fct
                NewFct(↓ name,↓ eType)
            ELSE                              — i.e. prev. forward
                IF name=nameOfMain THEN
                    SetMAINFctType (↓ eType, ↓ objOfMain,↓ opOfMain)
                ELSE
                    CheckTypes (↓ idList,↓ typeList,↓ len,
                                ↓ name,↓ op.type,↓ eTypeRet)
                END
            END
        END                                   — name in symbollist (currLevel)
( ==
        BeginFct(↓ name)
        FOR i := 1 TO len DO
            IF existInSymbolTable(idList[i]) THEN
                Rename(↓ idList[i],↓ CurrentLevel())
                idList[i] := GetVirtualName(↓ idList[i])
            END
```

END — rename parameters
SetParNames(↓ name,↓ idList,↓ len,↓ eType)
NewParList(↓ name,↓ idList,↓ typeList,↓ len)
Expression$_{\uparrow\text{ast}\uparrow\text{eTypeExpr}}$
CC: equalType(↓ eTypeRet,↓ eTypeExpr)
UnRenameLevel(↓ CurrentLevel())
EndFct(↓ name)
TypesToFlags (↓ typeList, ↓ len,↑ flagList)
MakeASTLambda(↓ idList,↓ flagList,↓ len,
↓ ast,↓ flag(eTypeExpr),↑ astLambda)
MakeASTDef(↓ name,↓ astLambda,↓ flag(eTypeExpr),↑ x)
IF fwd THEN — fct was forwarded
ResolveForwardFct(↓ name,↓ ast)
ELSE
AssignAST (↓ name,↓ ast)
END
|ε
MakeForwardFct(↓ name)
)

TCFDefinition$_{\downarrow\text{name}\uparrow\text{eType}\uparrow\text{x}}$ =
x := emptyAST
TypeDefinition$_{\downarrow\text{name}\uparrow\text{eType}}$
| *ConstantDefinition*$_{\downarrow\text{name}\uparrow\text{eType}\uparrow\text{x}}$
| *FunctionDefinition*$_{\downarrow\text{name}\uparrow\text{eType}\uparrow\text{x}}$

Definition$_{\uparrow\text{eType}\uparrow\text{x}}$ =
[*Ident*$_{\uparrow\text{name}}$
TCFDefinition$_{\downarrow\text{name}\uparrow\text{eType}\uparrow\text{x}}$
]

DefinitionSequence$_{\uparrow\text{x}}$ =
ast := emptyAST
i := 1
Definition$_{\uparrow\text{eType}\uparrow\text{ast}}$
IF ast<>emptyAST THEN
def[i] := ast
INC(i)
END
{ ;
ast := emptyAST
Definition$_{\uparrow\text{eType}\uparrow\text{ast}}$
CC: i<maxASTypeListLen
IF ast<>emptyAST THEN
def[i] := ast

```
        INC(i)
    END
}
    IF i>1 THEN
        MakeASTDefSeq(↓ def,↓ i − 1,↑ x)
    END
```

Expression

Expression$_{\uparrow x \uparrow eType}$ =
    eType := anyTy
    x := emptyAST
ListExpr$_{\uparrow x \uparrow eType}$

|{
    BeginBlock
DefinitionSequence$_{\uparrow ast1}$
} *Expression*$_{\uparrow ast2 \uparrow eType}$
    MakeASTLet(↓ ast1,↓ ast2,↓ flag(eType),↑ x)
    UnRenameLevel(↓ CurrentLevel())
    EndBlock()

|IF
Expression$_{\uparrow astHead \uparrow eTypeHead}$
    CC: equalType(↓ eTypeHead,↓ boolType)
THEN *Expression*$_{\uparrow astThen \uparrow eTypeThen}$
ELSE *Expression*$_{\uparrow astElse \uparrow eTypeElse}$
    CC: equalType(↓ eTypeThen,↓ eTypeElse)
    MaxType(↓ eTypeThen,↓ eTypeElse,↑ eType)
    MakeASTIf(↓ astHead,↓ astThen,↓ astElse,↓ flag(eType),↑ x)

|CASE
    elseBody.exist := FALSE
Expression$_{\uparrow astHead \uparrow eTypeHead}$
    CC: isUnionType(↓ eTypeHead)
{ *Field*$_{\uparrow idType}$
    IF existInSymbolTable(idType.name) THEN
        Rename(↓ idType.name,↓ CurrentLevel())
        renamed := TRUE
    END — if necessary rename new const.
    GetVariantType(↓ eTypeHead,↓ idType.type,↑ eType1,↑ nr)
    NewConst (↓ idType.name,↓ eType1)
} *Expression*$_{\uparrow ast \uparrow eType}$
    body[1].number := nr
    body[1].name := idType.name
    body[1].ast := ast
    i := 2

DeleteVar(↓ idType.name)
IF renamed THEN
UnRename(↓ idType.name)
renamed := FALSE
END
{ { *Field*$_{\uparrow \text{idType}}$
IF existInSymbolTable(idType.name) THEN
Rename(↓ idType.name,↓ CurrentLevel())
renamed := TRUE
END — if necessary rename new const.
GetVariantType(↓ eTypeHead,↓ idType.type,↑ eType1,↑ nr)
NewConst (↓ idType.name,↓ eType1)
} *Expression*$_{\uparrow \text{ast}\uparrow \text{eType0}}$
CC: equalType(↓ eType0,↓ eType)
MaxType(↓ eType,↓ eType0,↑ eType)
body[i].number := nr
body[i].name := idType.name
body[i].ast := ast
INC(i)
DeleteVar(↓ idType.name)
IF renamed THEN
UnRename(↓ idType.name)
renamed := FALSE
END
} [
ELSE *Expression*$_{\uparrow \text{ast}\uparrow \text{eType0}}$
CC: equalType(↓ eType0,↓ eType)
MaxType(↓ eType,↓ eType0,↑ eType)
elseBody.exist := TRUE
elseBody.ast := ast
]
MakeASTCase(↓ astHead,↓ body,↓ i − 1,↓ elseBody,↓ flag(eType),↑ x)

ListExpr$_{\uparrow \text{x}\uparrow \text{eType}}$ =
CompoundExpr$_{\uparrow \text{x}\uparrow \text{eType}}$
cells[1] := x
typeList[1] := eType
i := 2
{ : : *Factor*$_{\uparrow \text{ast}\uparrow \text{eType}}$
CC: i<maxASTypeListLen ∧ i<maxTypeListLen
cells[i] := ast
typeList[i] := eType
INC(i)
CC: $\forall 1 \leq j \leq i-3$:
equalType(typeList[j], typeList[$j+1$])∧
isListOf(typeList[$i-1$], typeList[1])
}

MakeASTConsList (↓ cells,↓ i − 2,↓ cells[i − 1],↑ x)

CompoundExpr$_{\uparrow x \uparrow eType}$ =
SimpleExpr$_{\uparrow ast1 \uparrow eType}$
x := ast1
[*Relation*$_{\uparrow rel}$
CC: equalType(↓ eType,↓ boolType) OR
equalType(↓ eType,↓ charType) OR
equalType(↓ eType,↓ intType)
SimpleExpr$_{\uparrow ast2 \uparrow eType2}$
CC: equalType(↓ eType,↓ eType2)
MakeASTRelation(↓ rel,↓ ast1,↓ ast2,↑ x)
]
eType := boolType

Relation$_{\uparrow rel}$ =
=
rel := eqRelTy
|**#**
rel := neRelTy
|**<**
rel := ltRelTy
|**<=**
rel := leRelTy
|**>**
rel := gtRelTy
|**>=**
rel := geRelTy

SimpleExpr$_{\uparrow x \uparrow eType}$ =
sign := FALSE
[+
|**-**
sign := TRUE
] *Term*$_{\uparrow ast \uparrow eType}$
CC: NOT sign OR equalType(↓ eType,↓ intType)
IF sign THEN
MakeASTSign(↓ ast,↑ x)
ELSE
x := ast
END
{*AddOp*$_{\uparrow add}$
Term$_{\uparrow ast \uparrow eType2}$
CC: CASE add OF
plusArTy, minusArTy : equalType(↓ eType2,↓ intType)

orArTy : equalType(↓ boolType)
END
CC: equalType(↓ eType,↓ eType2)
MakeASTArithm(↓ add,↓ x,↓ ast,↓ flag(eType),↑ ast0)
x := ast0
}

AddOp$_{\uparrow \text{add}}$ =
+
add := plusArTy
|-
add := minusArTy
||
add := orArTy

Term$_{\uparrow \text{x} \uparrow \text{eType}}$ =
Factor$_{\uparrow \text{x} \uparrow \text{eType}}$
{*MulOp*$_{\uparrow \text{mul}}$
Factor$_{\uparrow \text{ast1} \uparrow \text{eType2}}$
CC: CASE mul OF
timesArTy, divArTy,
modArTy : equalType(↓ intType)
andArTy : equalType(↓ boolType)
END
CC: equalType(↓ eType2,↓ eType)
MakeASTArithm(↓ mul,↓ x,↓ ast1,↑ ast0)
x := ast0
}

MulOp$_{\uparrow \text{mul}}$ =
*
mul := timesArTy
|/
mul := divArTy
|%
mul := modArTy
|&
mul := andArTy

Factor$_{\uparrow \text{x} \uparrow \text{eType}}$ =
|*Number*$_{\uparrow \text{nr}}$
MakeASTAtom(↓ numberAtom,↓ noCharVal,↓ value,↓ int,↑ x)
eType := intType
|*Character*$_{\uparrow \text{c}}$
MakeASTAtom(↓ charAtom,↓ c,↓ noCardVal,↓ int,↑ x)

eType := charType
$|\mathit{String}_{\uparrow\text{saddr}\uparrow\text{slen}}$
StringToList(↓ saddr,↓ slen,↑ x)
eType := strType
$|(\ \mathit{Expression}_{\uparrow\text{x}\uparrow\text{eType}}\ )$
$|\tilde{}\ \mathit{Factor}_{\uparrow\text{ast}\uparrow\text{eType}}$
MakeASTNot(↓ ast,↑ x)
$|\mathit{Designator}_{\uparrow\text{name}\uparrow\text{nr}\uparrow\text{eType}\uparrow\text{eTypeU}}$
x := emptyAST
isUnionFct := isUnionType(↓ eTypeU) AND isFctType(↓ eType)
IF isFctType(↓ eType) AND NOT isUnionFct THEN
MakeASTAtom(↓ idAtom,↓ noCharVal,↓ name,↓ flag(eType),↑ x)
END
isHigherOrder := isParOrConst(name) AND isFct(eType)
{
eType0 := eType
ast := x
$(\ \mathit{ArgumentList}_{\downarrow\text{ast}\downarrow\text{eType0}\downarrow\text{isUnionFct}\downarrow\text{isHigherOrder}\uparrow\text{x}}$
IF isFctType(↓ eType) AND NOT isUnionFct THEN
eType0 := eType
GetReturnType(↓ eType0,↑ eType)
ELSIF isUnionType(eTypeU) THEN
MakeASTAtom (↓ numberAtom,↓ noCharVal,↓ nr,↓ int,↑ nrAST)
asts[1] := nrAST
asts[2] := x
MakeASTTuple (↓ asts,↓ 2,↑ x)
END
isUnionFct := FALSE
isHigherOrder := TRUE
|
IF x=emptyAST THEN
MakeASTAtom(↓ idAtom,↓ noCharVal,↓ name,↓ flag(eType), ↑ x)
END
$\mathit{Selector}_{\downarrow\text{ast}\downarrow\text{eType0}\uparrow\text{x}\uparrow\text{eType}}$
isUnionFct := FALSE
isHigherOrder := TRUE
)}
$|\varepsilon$
IF x=emptyAST THEN
MakeASTAtom(↓ idAtom,↓ noCharVal,↓ name,↓ flag(eType), ↑ x)
END
IF isUnionType(↓ eTypeU) THEN
eType := eTypeU
END
IF isFctType(eType) AND NOT isParOrConst(↓ name) THEN
x0 := x
MakeASTCurry(↓ x0,↑ x)
END

```
Designator↑name↑nr↑eType↑eTypeU =
Ident↑name
            TestAdd(↓ name)
            FindObject(↓ name,↑ obj,↑ op)
            CC: obj<>noObject
            eType := op.type
            eTypeU := op.type
[ $
            CC: isUnion(↓ eType)
            DisableRenaming
Ident↑nameTy
            EnableRenaming
            eType0 := eType
            GetVariantType(↓ eType0,↓ nameTy,↑ eType,↑ nr)
]


ArgumentList↓ast↓eTypeIn↓isUnionFct↓isHigherOrder↑x =
            len := 1
            i := 1
[ [
            CC: NOT isUnionType(↓ eTypeIn)
            IF isFctType(↓ eTypeIn) AND NOT isUnionFct THEN
                GetParTypeList(↓ eTypeIn,↓ idList,↑ defTypeList,↑ len)
            ELSIF isTupleType(↓ eTypeIn) THEN
                GetTupleTypeList(↓ eTypeIn,↑ defTypeList,↑ len)
            ELSIF isListType(↓ eTypeIn) THEN
                GetListType(↓ etypeIn,↑ listType)
            END
Expression↑ast1↑eType1
            CC:    IF isListType(↓ eTypeIn) THEN
                       equalType(↓ listType,↓ eType1)
                   ELSIF isTupleType(↓ eTypeIn) OR
                           (isFctType(↓ eTypeIn) AND NOT isUnionFct) THEN
                       len>0 AND equalType(↓ defTypeList[1],↓ eType1)
                   ELSE
                       equalType(↓ eTypeIn,↓ eType1)
                   END
            arg[i] := ast1
            typeList[i] := eType1
            INC(i)
{ , Expression↑ast2↑eType2
            CC:    IF isListType(↓ eTypeIn) THEN
                       equalType(↓ listType,↓ eType2)
                   ELSIF isTupleType(↓ eTypeIn) OR
                           (isFctType(↓ eTypeIn) AND NOT isUnionFct) THEN
```

```
                len≥i AND equalType(↓ defTypeList[i],↓ eType2)
            END
        arg[i] := ast2
        typeList[i] := eType2
        INC(i)
} ] ]
        CC: isListType(↓ eTypeIn) OR len=i-1 OR (isUnionFct AND (i=2))))
        IF isFctType(↓ eTypeIn) AND NOT isUnionFct THEN
            GetReturnType (↓ eTypeIn,↑ eTypeOut)
            IF isHigherOrder THEN
                MakeASTApplication(↓ ast,↓ arg,↓ i − 1,↓ flag(eTypeOut),↑ x)
            ELSE
                MakeASTCall(↓ ast,↓ arg,↓ i − 1,↓ flag(eTypeOut),↑ x)
            END
        ELSIF isTupleType(↓ eTypeIn) THEN
            MakeASTTuple(↓ arg,↓ i − 1,↑ x)
        ELSIF isListType(↓ eTypeIn) THEN
            MakeASTList(↓ arg,↓ i − 1,↑ x)
        ELSIF isUnionFct THEN
            x := ast1
        ELSE
            x := ast1
        END
```

$Selector_{\downarrow\mathrm{ast}\downarrow\mathrm{eTypeIn}\uparrow\mathrm{x}\uparrow\mathrm{eTypeOut}}$ =

```
        DisableRenaming
```

. $Ident_{\uparrow\mathrm{name}}$

```
        EnableRenaming
        CC:   isTupleType(↓ eTypeIn) OR
              (isListType(↓ eTypeIn) AND
              (name=nameOfHEAD OR name=nameOfTAIL) )
        IF (name=nameOfHEAD OR name=nameOfTAIL) THEN
            IF name=nameOfHEAD THEN
                GetListType (↓ eTypeIn,↑ eTypeOut)
            ELSE
                eTypeOut := eTypeIn
            END
            MakeASTAtom (↓ idAtom,↓ noCharVal,↓ name,↓ flag(eTypeOut),↑ ast1)
            arg[1] := ast
            MakeASTCall (↓ ast1,↓ arg,↓ 1,↓ flag(eTypeOut),↑ x)
        ELSE
            GetFieldType(↓ eTypeIn, ↓ name,↑ nr,↑ eTypeOut)
            MakeASTSelection(↓ nr,↓ ast,↓ flag(eTypeOut), ↑ x)
        END
```

Program

$Program_{\uparrow x} = DefinitionSequence_{\uparrow x}$
MakeASTProgram ($\downarrow$ x,$\uparrow$ x)

Appendix B

Denotational Semantics of the Code Generation

This appendix gives a formal description of the code generation in the EVE compiler. For this description a *denotational semantics* is used. Due to the description in [Schmidt, 1986], a denotational semantics consists of three parts:

- *An abstract syntax definition of the source language.* Section B.1 gives a defintion of the abstract syntax of the EVE language. This abstract syntax is in the compiler represented by an abstract syntax tree (AST). This abstract syntax tree is an abstract data type with `EveAST` as its data encapsulation. The separation of the language constructs into the three syntax classes Definitions, Expressions and the Program is kept in the abstract syntax and it will also be kept in the other parts of the denotational semantics.

- *The semantic algebras.* All the relevant semantic algebras are described in Section B.2. The most important of these algebras cover the abstract data types of registers, labels, events and environments. For each semantic algebra the necessary procedures and their properties are specified.

- *The valuation function.* This function maps a program in the abstract syntax to ADAM assembler code. Actually, a valuation function for each of the three syntax classes will be given. This is also useful as each syntax class requires a different input and output for the definition of this valuation function. The names of these valuation functions are:

  - **D** for *Definitions*,
  - **E** for *Expressions*,
  - **P** for the *Program*.

  Note that these valuation functions can already be read as a functional program that implements the code generation of the EVE compiler.

  Section B.3, which describes the valuation functions, is split into several subsections. Each of these subsections covers the translation of one construct of the abstract syntax into assembler code. However, it does not suffice to describe just the generated assembler code. To keep the information about all semantic algebras up to date one must also perform some operations on these algebras. In general, the description of the translation of a construct is split into the following parts:

- The *names of the input tuple and of the result tuple* of the valuation function. Note that the variables in the input tuple are universally quantified whereas the variables in the result tuple are defined in the **where** expression. From this part already the structure of the resulting tuple can be seen.
- The actual ADAM *assembler code*. This part will always be the first component in the local definitions of the **where** construct. However, the assembler code does not only contain single assembler statements but also whole code sequences. Such ADAM assembler sequences are usually named $\mathbf{C}_x$, where x denotes the syntax construct that is evaluated.
- The next part in the description contains some *initializations*. Especially, new registers, labels and events have to be fetched via calls to the corresponding procedures of the semantic algebras.
- If the language construct contains subexpressions with *free variables*, the free variables that are the input for the whole expression *must be forked* such that they are available for the subexpressions. Hence, for each subexpression it must be decided which free variables are also used in other parts of the whole expression. If a variable is not only used in that subexpression a SPLIT has to be performed to make the variable available to both usages. This part is called Forking common free variables in the description of the valuation function.
- The next important part contains, if necessary, *recursive calls to the valuation function* in order to get code sequences for the subexpressions of the current expression. As these calls state the input/output behaviour of the subexpressions they are essential for the understanding of the generated ADAM assembler program. To emphasis the importance of this part these recursive calls have been put into $\boxed{\text{frameboxes}}$.
- The last part in the description of the evaluation function covers the splitting into parallel threads of control (e.g. the splitting of the trigger thread that must often be done at the beginning of a code block) and the synchronization of parallel threads. The former is done via calls to the functions *Split*, *SplitP* and $\overline{Split}$, the latter is done via calls to *SignalTree*. This part is called Splitting and Synchronization in the description of the valuation function.

Additionally, to the abstract syntax definition, the description of the semantic algebras and of the valuation function, which already define a denotational semantics, we also describe in Section B.4 the code that is generated for the predefined functions in the EVE language. For all these predefined functions inline code instead of function calls is generated.

Note that in this formal description of the code generation process some aspects will be discussed that have not been covered in Section 4.4.2. These aspects are:

- The forking of common free variables.
- The different handling of integer and pointer values in the following cases:

  - In the splitting of common free variables and
  - in the writing of these values into tuple fields.

- In the ADAM assembler code there is no RECEIVE instruction available as it was used in the description of the code generation in Section 4.4.2. However, one can simulate the semantics of the statement

$$\text{RECEIVE } r, \mathit{next}$$

by using the statement

$$\text{COPYI } r, r, \mathit{next}.$$

In the description of the valuation functions an abbreviation for a sequence of applications of the same functions is used. Hence, the expression

$$x_1, \ldots, x_n, \mathit{RegState}' \;=\; \mathit{NewReg}(\mathit{RegState})^n$$

is nothing but a shorthand for

$$\begin{aligned} (x_1, \mathit{RegState}_1) &= \mathit{NewReg}(\mathit{RegState}) \\ (x_2, \mathit{RegState}_2) &= \mathit{NewReg}(\mathit{RegState}_1) \\ &\vdots \\ (x_n, \mathit{RegState}') &= \mathit{NewReg}(\mathit{RegState}_{n-1}). \end{aligned}$$

B.1 Abstract Syntax

Definition Statements

ConstantDefinition	=	`defN` *Ident* [*Expression*] .
FunctionDefinition	=	`defN` *Ident* [(`lambdaN` (`lheadN` *Ident* { *Ident* }) (`lbodyN` *Expression*))] .
Definition	=	*ConstantDefinition* | *FunctionDefinition* .
DefinitionSequence	=	`defseqN` *Definition* { *Definition* } .

Expressions

Expression	=	*ListExpr* | `letN` *DefinitionSequence Expression* | `ifN` *Expression Expression Expression* | *ArithBoolExpr* | *CurriedExpr* | *BracketExpr* | *Atom* .
ListExpr	=	`composeN` *Expression Expression* | `nilN` .
ArithBoolExpr	=	*Relation Expression Expression* | *UnaryOp Expression* | *BinOp Expression Expression* .
Relation	=	`eqN` | `neN` | `ltN` | `leN` | `gtN` | `geN` .
UnaryOp	=	`notN` | `minusSignN` .
BinOp	=	`plusN` | `minusN` | `timesN` | `divN` | `modN` | `orN` | `andN` .
CurriedExpr	=	`curryN` *Ident* { *Ident* } .
BracketExpr	=	`applyN` *BracketExpr* { *Expression* } | `callN` *Ident* { *Expression* } | `selectN` *Number BracketExpr* | *Ident* .
Atom	=	*Ident* | *Number* | *Char* .
Ident	=	`idN` *identifier* .
Number	=	`numberN` *number* .
Char	=	`charN` *character* .

Program

Program = `programN` *DefinitionSequence* .

B.2 Semantic Algebras

B.2.1 *Variables* (atomic domain)

- Domain: *Variable*.

B.2.2 *Flags* for checking compound objects (atomic domain)

- Domain: $Flags = \{\mathsf{int}, \mathsf{pointer}, \mathsf{none}\}$
- Properties: $\mathsf{int} \neq \mathsf{pointer} \neq \mathsf{none}$

B.2.3 Handling of *Registers*

- Domains: *StateOfRegisters*, *Register*.
- Operations:

$$\begin{aligned} InitReg &: StateOfRegisters \\ NewReg &: StateOfRegisters \rightarrow (Register, StateOfRegisters) \end{aligned}$$

- Properties:

Typed Variables:
$RegState, RegState', rs_0 \ \in \ StateOfRegisters$
$r, r', r_0 \ \in \ Register$

$Distinct(r_0, rs_0)$,
where
$(r_0, rs_0) \ = \ NewReg(InitReg)$,
$Distinct(r, RegState) \ :\Leftrightarrow$
$r \neq r' \ \wedge \ Distinct(r, RegState') \wedge Distinct(r', RegState')$,
where $(r', RegState') \ = \ NewReg(RegState)$.

B.2.4 Handling of *Labels*

- Domains: *StateOfLabels*, *Label*.
- Operations:

$$StartLabel, EmptyLabel \ : \ Label$$
$$InitLabel \ : \ StateOfLabels$$
$$NewLabel \ : \ StateOfLabels \rightarrow (Label, StateOfLabels)$$

- Properties:

Typed Variables:
$LabelState, LabelState', ls_0 \ \in \ StateOfLabels$
$l, l', l_0 \ \in \ Label$

$Distinct(l_0, ls_0), Distinct(StartLabel, InitLabel)$,
$Distinct(EmptyLabel, InitLabel), StartLabel \ \neq \ EmptyLabel$,
where
$(l_0, ls_0) \ = \ NewLabel(InitLabel)$,

$Distinct(l, LabelState) \ :\Leftrightarrow$
$l \neq l' \ \wedge \ Distinct(l, LabelState') \wedge Distinct(l', LabelState')$,
where $(l', LabelState') \ = \ NewLabel(LabelState)$

B.2.5 Handling of *Events*

- Domains: *StateOfEvents*, *Event*.
- Operations:

$$InitEvent \ : \ StateOfEvents$$
$$NewEvent \ : \ StateOfEvents \rightarrow (Event, StateOfEvents)$$

- Properties:

Typed Variables:
$EventState, EventState', es_0 \in StateOfEvents$
$e, e', e_0 \in Event$

$Distinct(e_0, es_0)$,
where
$(e_0, es_0) = NewEvent(InitEvent)$,
$Distinct(e, EventState) \;:\Leftrightarrow$
$e \neq e' \wedge Distinct(e, EventState') \wedge Distinct(e', EventState')$,
where $(e', EventState') = NewEvent(EventState)$

B.2.6 Generated Code

- Domain: *Code* (ADAM Assembler Code).

B.2.7 Binding *Registers*, *Labels* and *Flags* to *Variables*

- Domain:

$$Environment : Variable \rightarrow (Register, Label, Flags)$$

- Operations: The usual operations on functions that are based on set theory. Furthermore:

$AddToEnv : Environment \rightarrow Variable \rightarrow Register \rightarrow Label \rightarrow$
$\quad Flags \rightarrow Environment$
$BindVar : Variable \rightarrow Label \rightarrow Environment \rightarrow Environment$
$lookupRegister : Environment \rightarrow Variable \rightarrow Register$
$lookupLabel : Environment \rightarrow Variable \rightarrow Label$
$lookupFlag : Environment \rightarrow Variable \rightarrow Flags$

- Properties:

Typed Variables:
$l \in Label, env \in Environment, r \in Register, v \in Variable, f \in Flags$

$\forall l, env, v, f$:
$lookupRegister\ (AddToEnv\ env\ v\ r\ l\ f)\ v = r$
$lookupRegister\ (AddToEnv\ env\ v'\ r\ l\ f)\ v = lookupRegister\ env\ v$, **if** $v \neq v'$
$lookupLabel\ (AddToEnv\ env\ v\ r\ l\ f)\ v = l$
$lookupLabel\ (AddToEnv\ env\ v'\ r\ l\ f)\ v = lookupLabel\ env\ v$, **if** $v \neq v'$
$lookupFlag\ (AddToEnv\ env\ v\ r\ l\ f)\ v = f$
$lookupFlag\ (AddToEnv\ env\ v'\ r\ l\ f)\ v = lookupFlag\ env\ v$, **if** $v \neq v'$

$\forall newLab, env, v$:
$lookupLabel\ (BindVar\ v\ newLab\ env)\ v = newLab$
$lookupLabel\ (BindVar\ v'\ newLab\ env)\ v = lookupLabel\ env\ v$, **if** $v \neq v'$

B.2.8 *State* of the resources that must be handled

- Domain:

$$State \ : \ (StateOfRegisters, StateOfLabels, StateOfEvents)$$

- Operations: The usual operations on tuples (like fst for selecting the first component).

B.2.9 The result of the valuation function

- Domain:

$$Result \ : \ Code$$

$$DefResult \ : \ (Code, Label, State, Environment)$$

$$ExprResult \ : \ (Code, Register, (Label \times Label), State)$$

- Operations: The usual operations on tuples.

B.3 Valuation Function

Some useful auxiliary functions

Splitting one thread in various other threads (*no pointer token*)

$$Split \ : \ Label \rightarrow Int \rightarrow StateOfLabels \rightarrow (Code, [Label], StateOfLabels)$$

$$Split\ l\ n\ LabelState \ = \ (\mathbf{C}, [l_1, \ldots, l_n], LabelState''')$$

where

$$\begin{array}{l} \mathbf{IF}\ n = 1 \\ \quad \mathbf{THEN}\ l_1 = l \\ \qquad\qquad LabelState''' = LabelState \\ \quad \mathbf{ELSE} \end{array}$$

$$\mathbf{C} = \left(\begin{array}{lll} l: & \mathrm{SPLIT} & l_1, l_2 \\ & \mathbf{C}_{split,l_1} & \\ & \mathbf{C}_{split,l_2} & \end{array} \right)$$

$$l_1, l_2, LabelState' = NewLabel(LabelState)^2$$

$$\begin{array}{l} (\mathbf{C}_{split,l_1}, [l_1, \ldots, l_{(n\ DIV\ 2)}], LabelState'') \ = \\ Split(l_1, (n\ DIV\ 2), LabelState') \\ (\mathbf{C}_{split,l_2}, [l_{(n\ DIV\ 2)+1}, \ldots, l_n], LabelState''') \ = \\ Split(l_2, n - (n\ DIV\ 2), LabelState'') \end{array}$$

END

Splitting one thread in various other threads (*with pointer token*)

$$SplitP \ : \ Label \rightarrow Register \rightarrow Int \rightarrow StateOfLabels \rightarrow (Code, [Label], StateOfLabels)$$

$$SplitP\ l\ r\ n\ LabelState \ = \ (\mathbf{C}, [l_1, \ldots, l_n], LabelState''')$$

where

IF $n = 1$
THEN $l_1 = l$
$LabelState''' = LabelState$
ELSE

$$\mathbf{C} = \begin{pmatrix} l: & \text{COPYP} & r, r, +1 \\ & \text{SPLIT} & l_1, l_2 \\ & \mathbf{C}_{split,l_1} & \\ & \mathbf{C}_{split,l_2} & \end{pmatrix}$$

$$l_1, l_2, LabelState' = NewLabel(LabelState)^2$$

$$\begin{array}{l} (\mathbf{C}_{split,l_1}, [l_1, \ldots, l_{(n\ DIV\ 2)}], LabelState'') \ = \\ SplitP(l_1, (n\ DIV\ 2), LabelState') \\ (\mathbf{C}_{split,l_2}, [l_{(n\ DIV\ 2)+1}, \ldots, l_n], LabelState''') \ = \\ SplitP(l_2, n - (n\ DIV\ 2), LabelState'') \end{array}$$

END

Splitting one thread in various other threads

$$\overline{Split} \;:\; Label \to Register \to Int \to Flags \to StateOfLabels \to (Code, [Label], StateOfLabels)$$

$$\overline{Split}\ l\ r\ n\ f\ LabelState \;=\; \begin{array}{ll} SplitP\ l\ r\ n\ LabelState & ,\textbf{if } f = pointer \\ Split\ l\ n\ LabelState & ,\textbf{otherwise} \end{array}$$

Merging various threads into one thread

$$\begin{array}{lll} SignalTree & : & [\ Label\ ] \to Int \to \\ & & StateOfLabels \to StateOfEvents \to \\ & & (Code, StateOfLabels, StateOfEvents, Label) \end{array}$$

$$SignalTree\ [l_1, \ldots, l_n]\ n\ LabelState\ EventState \;=\; (\mathbf{C}, LabelState'', EventState'', l)$$

where

$$\begin{array}{l} \textbf{IF } n = 1 \\ \quad \textbf{THEN } l = l_1 \\ \qquad\qquad EventState'' = EventState \\ \qquad\qquad LabelState'' = LabelState \\ \quad \textbf{ELSE} \end{array}$$

$$\mathbf{C} = \left(\begin{array}{lll} l_1 : & \mathrm{WAIT} & e_1, l'_1 \\ l_2 : & \mathrm{WAIT} & e_1, l'_1 \\ & \vdots & \\ l_{2k-1} : & \mathrm{WAIT} & e_k, l'_k \\ l_{2k} : & \mathrm{WAIT} & e_k, l'_k \\ & \vdots & \\ l_n : & \mathrm{WAIT} & e_{(n\ DIV\ 2)+1}, l'_{(n\ DIV\ 2)+1} \\ & \mathbf{C}_{SignalTree} & \end{array} \right)$$

$$\begin{array}{lll} e_1, \ldots, e_{(n\ DIV\ 2)+1}, EventState' & = & NewEvent(EventState)^{(n\ DIV\ 2)+1} \\ l_1, \ldots, l_{(n\ DIV\ 2)+1}, LabelState' & = & NewLabel(LabelState)^{(n\ DIV\ 2)+1} \end{array}$$

$$\begin{array}{l} (\mathbf{C}_{SignalTree}, LabelState'', EventState'', l) \;= \\ SignalTree\ [l'_1, \ldots, l_{(n\ DIV\ 2)+1}]\ (n\ DIV\ 2) + 1\ LabelState'\ EventState' \end{array}$$

END

B.3.1 Valuation Function of *Expression*

$$\mathbf{E} \ : \ Expression \rightarrow Label \rightarrow State \rightarrow Environment \rightarrow ExprResult$$

Constant

$$\text{const} \ \in \ \{ \ \texttt{numberN} \ c, \ \texttt{charN} \ c, \ \texttt{TRUE}, \ \texttt{FALSE} \ \}$$

$$\begin{array}{l} \mathbf{E} \ [\text{const}] \ trigger \ (RegState, LabelState, EventState) \ env \ = \\ (\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState) \end{array}$$

where

$$\mathbf{C} = \begin{pmatrix} trigger: & \text{CONST} & r, +2 \\ & c' & \\ & \text{SPLIT} & result, signal \end{pmatrix}$$

$$c' \ = \left\{ \begin{array}{ll} c & \textbf{, if} \ \text{const} \ = \ \texttt{numberN} \ c \\ \texttt{ORD}(c) & \textbf{, if} \ \text{const} \ = \ \texttt{charN} \ c \\ 1 & \textbf{, if} \ \text{const} \ = \ \texttt{TRUE} \\ 0 & \textbf{, if} \ \text{const} \ = \ \texttt{FALSE} \end{array} \right\}$$

$$\begin{array}{rcl} result, signal, LabelState' & = & NewLabel(LabelState)^2 \\ (r, RegState') & = & NewReg(RegState) \end{array}$$

Simple Variable

$$\begin{array}{l} \mathbf{E} \ [\texttt{idN} \ v] \ trigger \ (RegState, LabelState, EventState) \ env \ = \\ (\mathbf{C}, r, (l, trigger), (RegState, LabelState', EventState') \end{array}$$

where

$$\mathbf{C} = ()$$

$$\begin{array}{rcl} r & = & lookupRegister(env, v) \\ l & = & lookupLabel(env, v) \end{array}$$

Unary Operations

$$\mathbf{E}\ [\mathrm{op}\ E]\ \mathit{trigger}\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{env}\ =$$

$$(\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}_E, \mathit{LabelState}_E, \mathit{EventState}_E))$$

where

$$\mathbf{C} = \left(\begin{array}{lll} & \mathbf{C}_E & \\ \mathit{result}_E : & \mathrm{op} & r, r_E, \mathit{result} \end{array} \right)$$

$$\mathit{result}, \mathit{LabelStateSignal}'\ =\ \mathit{NewLabel}(\mathit{LabelState}_m)$$

$$(r, \mathit{RegState}')\ =\ \mathit{NewReg}(\mathit{RegState})$$

$$\begin{array}{l} (\mathbf{C}_E, r_E, (\mathit{result}_E, \mathit{signal}), (\mathit{RegState}_E, \mathit{LabelState}_E, \mathit{EventState}_E))\ = \\ \mathbf{E}\ [E]\ \mathit{trigger}\ (\mathit{RegState}', \mathit{LabelStateSignal}', \mathit{EventState})\ \mathit{env} \end{array}$$

Compound Arithmetical and Logical Expressions

$$\mathbf{E}\ [\mathrm{op}\ E_1\ E_2]\ \mathit{trigger}\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{env}\ =$$

$$(\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}_{E_2}, \mathit{LabelState}_{E_2}, \mathit{EventState}_{E_2}))$$

where

$$\mathbf{C} = \left(\begin{array}{lll} \mathit{trigger} : & \mathrm{SPLIT} & l_{E_1}, l_{E_2} \\ & & \\ & \mathbf{C}_1 & \\ & \vdots & \\ & \mathbf{C}_k & \\ & & \\ & \mathbf{C}_{E_1} & \\ & \mathbf{C}_{E_2} & \\ & & \\ \mathit{result}_{E_1} : & \mathrm{WAIT} & e_1, +2 \\ \mathit{result}_{E_2} : & \mathrm{WAIT} & e_1, +1 \\ & \mathrm{COPYI} & r, r_{E_1}, +1 \\ & \mathrm{op} & r, r_{E_2}, \mathit{result} \\ \mathit{signal}_{E_1} & \mathrm{WAIT} & e_2, \mathit{signal} \\ \mathit{signal}_{E_2} & \mathrm{WAIT} & e_2, \mathit{signal} \end{array} \right)$$

$$\begin{aligned} fV_1 &= freeVars(E_1) \\ flag_1 &= getFlag(E_1) \\ fV_2 &= freeVars(E_2) \\ flag_2 &= getFlag(E_2) \end{aligned}$$

$$l_{E_1}, l_{E_2}, LabelState_m = NewLabel(LabelState)^2$$

$$result, signal, LabelStateSignal = NewLabel(LabelState_m)^2$$

$$(r, RegStateR) = NewReg(RegState)$$

$$e_1, e_2, e_3, EventState_{e_3} = NewEvent(EventState)^3$$

— Forking common free variables —

$$[z_1, \ldots, z_k] = fV_1 \cap fV_2$$

$$(r_1, l_1, f_1) = (lookupRegister\ env\ z_1\ ,\ lookupLabel\ env\ z_1\ ,\ lookupFlag\ env\ z_1)$$

$$\vdots$$

$$(r_k, l_k, f_k) = (lookupRegister\ env\ z_k\ ,\ lookupLabel\ env\ z_k\ ,\ lookupFlag\ env\ z_k)$$

$$(\mathbf{C}_1, [l_{1,1}, l_{1,2}], LabelState_1) = \overline{Split}\ l_1\ 2\ LabelStateSignal$$

$$\vdots$$

$$(\mathbf{C}_k, [l_{k,1}, l_{k,2}], LabelState_k) = \overline{Split}\ l_k\ 2\ LabelState_k$$

$$\begin{aligned} env_{E_1} &= BindVar(z_1, l_{1,1}) \circ \cdots \circ BindVar(z_k, l_{k,1})\ env \\ env_{E_2} &= BindVar(z_1, l_{1,2}) \circ \cdots \circ BindVar(z_k, l_{k,2})\ env \end{aligned}$$

— End of forking common free variables —

$$\boxed{\begin{array}{l} (\mathbf{C}_{E_1}, r_{E_1}, (result_{E_1}, signal_{E_1}), (RegState_{E_1}, LabelState_{E_1}, EventState_{E_1})) = \\ \mathbf{E}\ [E_1]\ l_{E_1}\ (RegStateR, LabelStateSignal, EventState_{e_2})\ env_{E_1} \\ \\ (\mathbf{C}_{E_2}, r_{E_2}, (result_{E_2}, signal_{E_2}), (RegState_{E_2}, LabelState_{E_2}, EventState_{E_2})) = \\ \mathbf{E}\ [E_2]\ l_{E_2}\ (RegState_{E_1}, LabelState_{E_1}, EventState_{E_1})\ env_{E_2} \end{array}}$$

Function Calls

$$\mathbf{E}\ [\ \texttt{callN}\ (\texttt{idN}\ f)\ E_1, \ldots, E_n\ ]\ \mathit{trigger}\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{env}\ =$$
$$(\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}_{E_n}, \mathit{LabelState}', \mathit{EventState}'))$$

where

$$\mathbf{C}\ =\ \left(\begin{array}{lll}
 & \mathbf{C}_{split} & \\
 & & \\
\mathit{lsplit}_0 : & \text{FASTCALL} & f, +1 \\
 & \text{COPYI} & r_{act}, r_{act}, \mathit{send} \\
 & \text{COPYI} & r, r, \mathit{result} \\
 & & \\
 & \mathbf{C}_{send} & \\
 & & \\
 & \mathbf{C}_{1,1} & \\
 & \vdots & \\
 & \mathbf{C}_{1,m_1} & \\
 & & \\
 & \mathbf{C}_{E_1} & \\
 & & \\
\mathit{result}_{E_1} : & \text{WAIT} & e_1, +2 \\
\mathit{send}_1 : & \text{WAIT} & e_1, +1 \\
 & \text{SEND} & r_{E_1}, r_{act}, 1 \\
 & \text{JUMP} & \mathit{arg}_1 \\
 & & \\
 & \vdots & \\
 & & \\
 & \mathbf{C}_{n-1,1} & \\
 & \vdots & \\
 & \mathbf{C}_{n-1,m_{n-1}} & \\
 & & \\
 & \mathbf{C}_{E_{n-1}} & \\
 & & \\
\mathit{result}_{E_{n-1}} : & \text{WAIT} & e_{n-1}, +2 \\
\mathit{send}_{n-1} : & \text{WAIT} & e_{n-1}, +1 \\
 & \text{SEND} & r_{E_{n-1}}, r_{act}, n-1 \\
 & \text{JUMP} & \mathit{arg}_{n-1} \\
 & & \\
 & \mathbf{C}_{E_n} & \\
 & & \\
\mathit{result}_{E_n} : & \text{WAIT} & e_n, +2 \\
\mathit{send}_n : & \text{WAIT} & e_n, +1 \\
 & \text{SEND} & r_{E_n}, r_{act}, n \\
 & \text{JUMP} & \mathit{arg}_n \\
 & & \\
 & \mathbf{C}_{ST} &
\end{array}\right)$$

$$\begin{aligned} fV_1 &= freeVars(E_1) \\ &\vdots \\ fV_n &= freeVars(E_n) \end{aligned}$$

$$r_{act}, r, RegState_{result} = NewReg(RegState)^2$$

$$\begin{aligned} &send, result, signal, \\ &arg_1, \ldots, arg_n, LabelState_{signal} = NewLabel(LabelState)^{n+3} \end{aligned}$$

$$e_1, \ldots, e_n, EventState' = NewEvent(EventState)^n$$

— Forking common free variables —

$[y_{1,1}, \ldots, y_{1,m_1}] \;=\; fV_1 \;\cap (fV_2 \cup \cdots \cup fV_n)$
$(ry_{1,1}, ly_{1,1}, fy_{1,1}) \;=$
$(lookupRegister\ env\ y_{1,1}\ ,\ lookupLabel\ env\ y_{1,1}\ ,\ lookupFlag\ env\ y_{1,1})$

$$\vdots$$

$(ry_{1,m_1}, ly_{1,m_1}, fy_{1,m_1}) \;=$
$(lookupRegister\ env\ y_{1,m_1}\ ,\ lookupLabel\ env\ y_{1,m_1}\ ,\ lookupFlag\ env\ y_{1,m_1})$

$(\mathbf{C}_{1,1}, [l_{1,1}, l'_{1,1}], LabelState_{1,1}) \;= \overline{Split}(ly_{1,1}, ry_{1,1}, 2, fy_{1,1}, LabelState_{signal})$

$$\vdots$$

$(\mathbf{C}_{1,m_1}, [l_{1,m_1}, l'_{1,m_1}], LabelState_{1,m_1}) \;= \overline{Split}(ly_{1,m_1}, ry_{1,m_1}, 2, fy_{1,m_1},$
$LabelState_{1,m_1-1})$

$env_{E_1} \;=\; BindVar\ (y_{1,1}\ l_{1,1})\ \circ \cdots \circ (y_{1,m_1}\ l_{1,m_1})\ env$
$env'_{E_1} \;=\; BindVar\ (y_{1,1}\ l'_{1,1})\ \circ \cdots \circ (y_{1,m_1}\ l'_{1,m_1})\ env$

$$\vdots$$

$[y_{n-1,1}, \ldots, y_{n-1,m_{n-1}}] \;=\; fV_{n-1} \;\cap fV_n$
$(ry_{n-1,1}, ly_{n-1,1}, fly_{n-1,1}) \;=$
$(lookupRegister\ env'_{E_{n-2}}\ y_{n-1,1}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n-1,1}\ ,$
$lookupFlag\ env'_{E_{n-2}}\ y_{n-1,1})$

$$\vdots$$

$(ry_{n-1,m_{n-1}}, ly_{n-1,m_{n-1}}, fy_{n-1,m_{n-1}}) \;=$
$(lookupRegister\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}}\ ,$
$lookupFlag\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}})$

$(\mathbf{C}_{n-1,1}, [l_{n-1,1}, l'_{n-1,1}], LabelState_{n-1,1}) \;=$
$\overline{Split}(ly_{n-1,1}, ry_{n-1,1}, 2, fy_{n-1,1}, LabelState_{n-2,m_{n-2}})$

$$\vdots$$

$(\mathbf{C}_{n-1,m_{n-1}}, [l_{n-1,m_{n-1}}, l'_{n-1,m_{n-1}}], LabelState_{n-1,m_{n-1}}) \;=$
$\overline{Split}(ly_{n-1,m_{n-1}}, ry_{n-1,m_{n-1}}, 2, fy_{n-1,m_{n-1}}, LabelState_{n-1,m_{n-1}-1})$

$env_{E_{n-1}} \;=\; BindVar\ (y_{n-1,1}\ l_{n-1,1})\ \circ \cdots \circ (y_{n-1,m_{n-1}}\ l_{n-1,m_{n-1}})\ env'_{E_{n-2}}$
$env'_{E_{n-1}} \;=\; BindVar\ (y_{n-1,1}\ l'_{n-1,1})\ \circ \cdots \circ (y_{n-1,m_{n-1}}\ l'_{n-1,m_{n-1}})\ env'_{E_{n-2}}$

— End of forking common free variables —

$$(\mathbf{C}_{E_1}, r_{E_1}, (result_{E_1}, signal_{E_1}), (RegState_{E_1}, LabelState_{E_1}, EventState_{E_1})) \ = \ \mathbf{E}\ [E_1]\ l_{split_1}\ (RegState_{result}, LabelState_{signal}, EventState_{e_n})\ env_{E_1}$$

$$(\mathbf{C}_{E_{n-1}}, r_{E_{n-1}}, (result_{E_{n-1}}, signal_{E_{n-1}}), (RegState_{E_{n-1}}, LabelState_{E_{n-1}}, EventState_{E_{n-1}})) \ = \ \mathbf{E}\ [E_{n-1}]\ l_{split_{n-1}}\ (RegState_{E_{n-2}}, LabelState_{E_{n-2}}, EventState_{E_{n-2}})\ env_{E_{n-1}}$$

$$(\mathbf{C}_{E_n}, r_{E_n}, (result_{E_n}, signal_{E_n}), (RegState_{E_n}, LabelState_{E_n}, EventState_{E_n})) \ = \ \mathbf{E}\ [E_n]\ l_{split_n}\ (RegState_{E_{n-1}}, LabelState_{E_{n-1}}, EventState_{E_{n-1}})\ env_{E_n}$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{split}, [l_{split_0}, \ldots, l_{split_n}], LabelState_{split}) \ = \ Split(trigger, n+1, LabelState_{E_n})$$

$$(\mathbf{C}_{send}, [send_1, \ldots, send_n], LabelState_{send}) \ = \ Split(send, n, LabelState_{split})$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \ = \ SignalTree([arg_1, \ldots, arg_n, signal_{E_1}, \ldots, signal_{E_n}], 2n, LabelState_{send}, EventState_{E_n})$$

— End of Splitting and Synchronizing —

Conditional Expression

$$\mathbf{E}\ [\ \texttt{ifN}\ E_1\ E_2\ E_3\ ]\ trigger\ (RegState, LabelState, EventState)\ env \ = \ (\mathbf{C}, r, (result, signal), (RegState\text{-}r_{E_3}, LabelState_{ST_{error}}, EventState_{ST_{error}}))$$

$$
\mathbf{C} = \left(\begin{array}{lll}
trigger: & \mathrm{SPLIT} & trigger_{E_1}, trigger_{E_{23}} \\
\\
 & \mathbf{C}_{1,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{1,m} & \\
\\
 & \mathbf{C}_{E_1} & \\
\\
 & \mathbf{C}_{split} & \\
\\
signal_{E_1}: & \mathrm{WAIT} & e, +2 \\
head: & \mathrm{WAIT} & e, +1 \\
 & \mathrm{SWITCH} & r_{E_1}, lST_{true}, lST_{false} \\
 & \mathrm{JUMP} & l_{error,0} \\
\\
trigger_{E_{23}}: & \mathrm{WAIT} & e_0, +2 \\
trigger_0: & \mathrm{WAIT} & e_0, +1 \\
 & \mathrm{SWITCH} & r_{E_1}, l_{true}, l_{false} \\
 & \mathrm{COPYI} & r, r_{E_1}, result \\
\\
ly_1: & \mathrm{WAIT} & e_1, +2 \\
trigger_1: & \mathrm{WAIT} & e_1, +1 \\
 & \mathrm{SWITCH} & r_{E_1}, l_{true,1}, l_{false,1} \\
 & \mathrm{JUMP} & l_{error,1} \\
 & \quad\vdots & \\
ly_n: & \mathrm{WAIT} & e_n, +2 \\
trigger_n: & \mathrm{WAIT} & e_n, +1 \\
 & \mathrm{SWITCH} & r_{E_n}, l_{true,n}, l_{false,n} \\
 & \mathrm{JUMP} & l_{error,n} \\
\\
 & \mathbf{C}_{E_2} & \\
 & \mathbf{C}_{ST_{E_2}} & \\
\\
 & \mathbf{C}_{E_3} & \\
 & \mathbf{C}_{ST_{E_3}} & \\
\\
 & \mathbf{C}_{ST_{error}} & \\
\\
result_{E_2}: & \mathrm{COPYI} & r, r_{E_2}, +1 \\
 & \mathrm{JUMP} & result \\
result_{E_3}: & \mathrm{COPYI} & r, r_{E_3}, +1 \\
 & \mathrm{JUMP} & result \\
\\
signal'_{E_2}: & \mathrm{JUMP} & signal \\
signal'_{E_3}: & \mathrm{JUMP} & signal
\end{array}\right)
$$

where

$$\begin{aligned} fV_1 &= freeVars(E_1) \\ fV_2 &= freeVars(E_2) \\ fV_3 &= freeVars(E_3) \end{aligned}$$

$$\begin{aligned} &trigger_{E_1}, trigger_{E_{23}}, head, trigger_1, \ldots, trigger_n, lST_{true}, l_{true}, l_{true,1}, \ldots, l_{true,n}, \\ &lST_{false}, l_{false}, l_{false,1}, \ldots, l_{false,n}, l_{error,0}, l_{error,1}, \ldots, l_{error,n}, \\ &signal'_{E_2}, signal'_{E_3}, result, signal, LabelStateLFalse_n \;=\; NewLabel(LabelState)^{3n+11} \end{aligned}$$

$$e, e_0, e_1, \ldots, e_n, EventStateE_n \;=\; NewEvent(EventStateE)^{n+2}$$

$$(r, RegStateR) \;=\; NewReg(RegState)$$

— Forking common free variables —

$$[x_1, \ldots, x_m] \;=\; fV_1 \cap (fV_2 \cup fV_3)$$

$$(rx_1, lx_1, fx_1) \;=\; (lookupRegister\ env\ x_1\ ,\ lookupLabel\ env\ x_1\ ,\ lookupFlag\ env\ x_1)$$

$$\vdots$$

$$(rx_m, lx_m, fx_m) \;=\; (lookupRegister\ env\ x_m\ ,\ lookupLabel\ env\ x_m\ ,\ lookupFlag\ env\ x_m)$$

$$(\mathbf{C}_{1,1}, [l_1, l'_1], LabelState_{1,1}) \;=\; \overline{Split}(lx_1, rx_1, 2, fx_1, LabelStateLFalse_n)$$

$$\vdots$$

$$(\mathbf{C}_{1,m}, [l_m, l'_m], LabelState_{1,m}) \;=\; \overline{Split}(lx_m, rx_m, 2, fx_m, LabelState_{1,m-1})$$

$$\begin{aligned} env_{E_1} &= BindVar\ (x_1\ l_1)\ \circ \cdots \circ (x_m\ l_m)\ env \\ env'_{E_1} &= BindVar\ (x_1\ l'_1)\ \circ \cdots \circ (x_m\ l'_m)\ env \end{aligned}$$

— End of forking common free variables —

$$[y_1, \ldots, y_n] = fV_2 \cup fV_3$$

$$ly_1 = lookupLabel\ env\ y_1$$
$$\vdots$$
$$ly_n = lookupLabel\ env\ y_n$$

$$I_2 = \{j \mid y_j \in fV_2\}$$
$$I_2' = \{1, \ldots, n\} - I_2$$
$$I_3 = \{j \mid y_j \in fV_3\}$$
$$I_3' = \{1, \ldots, n\} - I_3$$

$$env_{E_2} = (\bigcirc_{j \in I_2} BindVar\ (y_j\ l_{true,j}))\ env'_{E_1}$$
$$st_{E_2} = [\ l_{true,j} \mid j \in I_2'\ ]$$

$$env_{E_3} = (\bigcirc_{j \in I_3} BindVar\ (y_j\ l_{false,j}))\ env'_{E_1}$$
$$st_{E_3} = [\ l_{false,j} \mid j \in I_3'\ ]$$

$$(\mathbf{C}_{E_1}, r_{E_1}, (result_{E_1}, signal_{E_1}), (RegState', LabelState', EventState')) =$$
$$\mathbf{E}\ [E_1]\ trigger_{E_1}\ (RegStateR, LabelStateL_{False,n}, EventState_{e_n})\ env_{E_1}$$

$$(\mathbf{C}_{E_2}, r_{E_2}, (result_{E_2}, signal_{E_2}), (RegState'', LabelState'', EventState'')) =$$
$$\mathbf{E}\ [E_2]\ l_{true}\ (RegState', LabelState', EventState')\ env_{E_2}$$

$$(\mathbf{C}_{E_3}, r_{E_3}, (result_{E_3}, signal_{E_3}), (RegState''', LabelState''', EventState''')) =$$
$$\mathbf{E}\ [E_3]\ l_{false}\ (RegState'', LabelState'', EventState'')\ env_{E_3}$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{split}, [head, trigger_0, trigger_1, \ldots, trigger_n], LabelState'''') =$$
$$Split(result_{E_1}, n+2, LabelState_{ST_{E_3}})$$

$$(\mathbf{C}_{ST_{E_2}}, LabelState_{ST_{E_2}}, EventState_{ST_{E_2}}, signal'_{E_2}) =$$
$$SignalTree(lST_{true}\ :\ signal_{E_2}\ :\ st_{E_2}, |I_2|+2, LabelState'', EventState'')$$

$$(\mathbf{C}_{ST_{E_3}}, LabelState_{ST_{E_3}}, EventState_{ST_{E_3}}, signal'_{E_3}) =$$
$$SignalTree(lST_{false}\ :\ signal_{E_3}\ :\ st_{E_3}, |I_3|+2, LabelState''', EventState''')$$
$$(\mathbf{C}_{ST_{error}}, LabelState_{ST_{error}}, EventState_{ST_{error}}, signal) =$$
$$SignalTree([l_{error,0}, l_{error,1}, \ldots, l_{error,n}], n+1, LabelState'''', EventState_{ST_{E_3}})$$

— End of Splitting and Synchronizing —

Local Definitions

$$\mathbf{E}\,[\ \texttt{letN}\ D\ E\ ]\ trigger\ (RegState, LabelState, EventState)\ env\ =\ (\mathbf{C}, r, (result, signal), (RegState_E, LabelState', EventState'))$$

where

$$\mathbf{C}\ =\ \left(\begin{array}{lll} trigger: & \text{SPLIT} & trigger_D, trigger_E \\ & \mathbf{C}_1 & \\ & \vdots & \\ & \mathbf{C}_m & \\ & \mathbf{C}_D & \\ & \mathbf{C}_E & \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$trigger_D, trigger_E, send, LabelState_0\ =\ NewLabel(LabelState)^3$$

$$\begin{aligned} fV_1 &= freeVars(D) \\ fV_2 &= freeVars(E) \end{aligned}$$

— Forking common free variables —

$$[x_1, \ldots, x_m]\ =\ fV_1\ \cap\ fV_2$$

$$(r_1, l_1, f_1)\ =\ (lookupRegister\ env\ x_1\ ,\ lookupLabel\ env\ x_1\ ,\ lookupFlag\ env\ x_1)$$

$$\vdots$$

$$(r_m, l_m, f_m)\ =\ (lookupRegister\ env\ x_m\ ,\ lookupLabel\ env\ x_m\ ,\ lookupFlag\ env\ x_m)$$

$$(\mathbf{C}_1, [l_{1,D}, l_{1,E}], LabelState_1)\ =\ \overline{Split}(l_1, r_1, 2, f_1, LabelState_0)$$

$$\vdots$$

$$(\mathbf{C}_m, [l_{m,D}, l_{m,E}], LabelState_m)\ =\ \overline{Split}(l_m, r_m, 2, f_m, LabelState_{m-1})$$

$$\begin{aligned} env_D &= BindVar(x_1\ l_{1,D})\ \circ \cdots \circ BindVar(x_m\ l_{m,D})\ env \\ env_E &= BindVar(x_1\ l_{1,E})\ \circ \cdots \circ BindVar(x_m\ l_{m,E})\ env'_D \end{aligned}$$

— End of forking common free variables —

$$(\mathbf{C}_D, signal_D, (RegisterState_D, LabelState_D, EventState_D), env'_D) \;=\; \mathbf{D}\;[D]\; trigger_D\; (RegisterState', LabelState_m, EventState)\; fV_2\; env_D$$

$$(\mathbf{C}_E, r, (result, signal), (RegisterState', LabelState_E, EventState_E)) \;=\; \mathbf{E}\;[E]\; trigger_E\; (RegisterState_D, LabelState_D, EventState_D)\; env_E$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \;=\; SignalTree([signal_D, signal_E], 2, LabelState_E, EventState_E)$$

— End of Splitting and Synchronizing —

Empty Tuple

$$\mathbf{E}\;[\;\texttt{nilN}\;]\; trigger\; (RegState, LabelState, EventState)\; env \;=\; (\mathbf{C}, r, (result, signal), (RegState_{E_n}, LabelState', EventState'))$$

where

$$\mathbf{C} \;=\; \left(\begin{array}{lll} trigger: & \mathrm{CONST} & r, +2 \\ & 0 & \\ & \mathrm{JUMP} & result, signal \end{array}\right)$$

$$(r, RegState') \;=\; NewReg(RegState)$$

$$result, signal, LabelState' \;=\; NewLabel(LabelState)^2$$

Tuple Construction

$$\mathbf{E}\;[\;\texttt{composeN}\; E_1\; \ldots\; E_n\;]\; trigger\; (RegState, LabelState, EventState)\; env \;=\; (\mathbf{C}, r, (result, signal), (RegState_{E_n}, LabelState', EventState'))$$

where

$$
\mathbf{C} = \left(\begin{array}{lll}
\mathit{trigger}: & \mathrm{ALLOC} & n, +1 \\
 & \mathrm{COPYP} & r, r, +1 \\
 & \mathrm{SPLIT} & \mathit{result}, l_{\mathit{split}} \\
 & & \\
 & \mathbf{C}_{splitP} & \\
 & & \\
 & \mathbf{C}_{1,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{1,m_1} & \\
 & & \\
l_{\mathit{split}_{E_1}}: & \mathrm{COPYP} & r, r, +1 \\
 & \mathrm{SPLIT} & \mathit{trigger}_{E_1}, \mathit{gc}_{E_1} \\
 & \mathbf{C}_{E_1} & \\
l'_{\mathit{split}_{E_1}}: & \mathrm{WAIT} & e_{E_1}, +2 \\
\mathit{result}_{E_1}: & \mathrm{WAIT} & e_{E_1}, +1 \\
\mathit{gc}_{E_1}: & \mathrm{WAIT} & e'_{E_1}, \mathit{result}'_{E_1} \\
 & \mathbf{C}'_{E_1} & \\
 & \mathrm{JUMP} & \mathit{end}_1 \\
 & & \\
 & \quad\vdots & \\
 & & \\
 & \mathbf{C}_{n-1,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{n-1,m_{n-1}} & \\
 & & \\
l_{\mathit{split}_{E_{n-1}}}: & \mathrm{COPYP} & r, r, +1 \\
 & \mathrm{SPLIT} & \mathit{trigger}_{E_{n-1}}, \mathit{gc}_{E_{n-1}} \\
 & \mathbf{C}_{E_{n-1}} & \\
l'_{\mathit{split}_{E_{n-1}}}: & \mathrm{WAIT} & e_{E_{n-1}}, +2 \\
\mathit{result}_{E_{n-1}}: & \mathrm{WAIT} & e_{E_{n-1}}, +1 \\
\mathit{gc}_{E_{n-1}}: & \mathrm{WAIT} & e'_{E_{n-1}}, \mathit{result}'_{E_{n-1}} \\
 & \mathbf{C}'_{E_{n-1}} & \\
 & \mathrm{JUMP} & \mathit{end}_{n-1} \\
 & & \\
 & \mathbf{C}_{E_n} & \\
l'_{\mathit{split}_{E_n}}: & \mathrm{WAIT} & e_{E_n}, +2 \\
\mathit{result}_{E_n}: & \mathrm{WAIT} & e_{E_n}, +1 \\
\mathit{gc}_{E_n}: & \mathrm{WAIT} & e'_{E_n}, \mathit{result}'_{E_n} \\
 & \mathbf{C}'_{E_n} & \\
 & \mathrm{JUMP} & \mathit{end}_n \\
 & & \\
 & \mathbf{C}_{ST} &
\end{array} \right)
$$

$$\mathbf{C}'_{E_1} = \begin{cases} \left(\begin{array}{lll} \mathit{result}'_{E_1}: & \text{WRITEI} & r_1, r, 0 \end{array} \right) & \textbf{if } \mathit{flag}_1 = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}'_{E_1}: & \text{COPYP} & r_1, r_1, +1 \\ & \text{WRITEP} & r_1, r, 0 \end{array} \right) & \textbf{otherwise} \end{cases}$$

$$\vdots$$

$$\mathbf{C}'_{E_{n-1}} = \begin{cases} \left(\begin{array}{lll} \mathit{result}'_{E_{n-1}}: & \text{WRITEI} & r_{n-1}, r, n-2 \end{array} \right) & \textbf{if } \mathit{flag}_{n-1} = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}'_{E_{n-1}}: & \text{COPYP} & r_{n-1}, r_{n-1}, +1 \\ & \text{WRITEP} & r_{n-1}, r, n-2 \end{array} \right) & \textbf{otherwise} \end{cases}$$

$$\mathbf{C}'_{E_n} = \begin{cases} \left(\begin{array}{lll} \mathit{result}'_{E_n}: & \text{WRITEI} & r_n, r, n-1 \end{array} \right) & \textbf{if } \mathit{flag}_n = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}'_{E_n}: & \text{COPYP} & r_n, r_n, +1 \\ & \text{WRITEP} & r_n, r, n-1 \end{array} \right) & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} fV_1 &= \mathit{freeVars}(E_1) \\ &\vdots \\ fV_n &= \mathit{freeVars}(E_n) \\ \\ \mathit{flag}_1 &= \mathit{getFlag}(E_1) \\ &\vdots \\ \mathit{flag}_n &= \mathit{getFlag}(E_n) \end{aligned}$$

$(r, \mathit{RegStateR}) = \mathit{NewReg}(\mathit{RegState})$
$\mathit{result}, \mathit{signal}, \mathit{LabelStateRS} = \mathit{NewLabel}(\mathit{LabelState})^2$

$l_{split}, \mathit{result}'_{E_1}, \ldots, \mathit{result}'_{E_n}, gc_{E_1}, \ldots, gc_{E_1},$
$\mathit{trigger}_{E_1}, \ldots, \mathit{trigger}_{E_{n-1}}, \mathit{end}_1, \ldots, \mathit{end}_n, \mathit{LabelStateTrigger} =$
$\mathit{NewLabel}(\mathit{LabelStateRS})^{4n}$

$e_{E_1}, \ldots, e_{E_n}, e'_{E_1}, \ldots, e'_{E_n}, \mathit{EventState}' = \mathit{NewEvent}(\mathit{EventState})^{2n}$

— Forking common free variables —

$$[y_{1,1},\ldots,y_{1,m_1}] \;=\; fV_1 \;\cap (fV_2 \cup \cdots \cup fV_n)$$
$$(ry_{1,1}, ly_{1,1}, fy_{1,1}) \;=$$
$$(lookupRegister\ env\ y_{1,1}\ ,\ lookupLabel\ env\ y_{1,1}\ ,$$
$$lookupFlag\ env\ y_{1,1})$$
$$\vdots$$
$$(ry_{1,m_1}, ly_{1,m_1}, fy_{1,m_1}) \;=$$
$$(lookupRegister\ env\ y_{1,m_1}\ ,\ lookupLabel\ env\ y_{1,m_1}\ ,$$
$$lookupFlag\ env\ y_{1,m_1})$$

$$(\mathbf{C}_{1,1}, [l_{1,1}, l'_{1,1}], 2, LabelState_{1,1}) \;= \overline{Split}(ly_{1,1}, ry_{1,1}, 2, fy_{1,1}, LabelState_{signal})$$
$$\vdots$$
$$(\mathbf{C}_{1,m_1}, [l_{1,m_1}, l'_{1,m_1}], 2, LabelState_{1,m_1}) \;=$$
$$\overline{Split}(ly_{1,m_1}, ry_{1,m_1}, 2, fy_{1,m_1}, LabelState_{1,m_1-1})$$

$$env_{E_1} \;=\; BindVar\ (y_{1,1}\ l_{1,1})\ \circ\cdots\circ (y_{1,m_1}\ l_{1,m_1})\ env$$
$$env'_{E_1} \;=\; BindVar\ (y_{1,1}\ l'_{1,1})\ \circ\cdots\circ (y_{1,m_1}\ l'_{1,m_1})\ env$$

$$\vdots$$

$$[y_{n-1,1},\ldots,y_{n-1,m_{n-1}}] \;=\; fV_{n-1} \;\cap fV_n$$
$$(ry_{n-1,1}, ly_{n-1,1}, fy_{n-1,1}) \;=$$
$$(lookupRegister\ env'_{E_{n-2}}\ y_{n-1,1}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n-1,1}\ ,$$
$$lookupFlag\ env'_{E_{n-2}}\ y_{n-1,1})$$
$$\vdots$$
$$(ry_{n-1,m_{n-1}}, ly_{n-1,m_{n-1}}, ly_{n-1,m_{n-1}}) \;=$$
$$(lookupRegister\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}}\ ,$$
$$lookupFlag\ env'_{E_{n-2}}\ y_{n-1,m_{n-1}})$$

$$(\mathbf{C}_{n-1,1}, [l_{n-1,1}, l'_{n-1,1}], LabelState_{n-1,1}) \;=$$
$$\overline{Split}(ly_{n-1,1}, ry_{n-1,1}, 2, fy_{n-1,1}, LabelState_{n-2,m_{n-2}})$$
$$\vdots$$
$$(\mathbf{C}_{n-1,m_{n-1}}, [l_{n-1,m_{n-1}}, l'_{n-1,m_{n-1}}], LabelState_{n-1,m_{n-1}}) \;=$$
$$\overline{Split}(ly_{n-1,m_{n-1}}, ry_{n-1,m_{n-1}}, 2, fy_{n-1,m_{n-1}}, LabelState_{n-1,m_{n-1}-1})$$

$$env_{E_{n-1}} \;=\; BindVar\ (y_{n-1,1}\ l_{n-1,1})\ \circ\cdots\circ (y_{n-1,m_{n-1}}\ l_{n-1,m_{n-1}})\ env'_{E_{n-2}}$$
$$env'_{E_{n-1}} \;=\; BindVar\ (y_{n-1,1}\ l'_{n-1,1})\ \circ\cdots\circ (y_{n-1,m_{n-1}}\ l'_{n-1,m_{n-1}})\ env'_{E_{n-2}}$$

— End of forking common free variables —

$$(\mathbf{C}_{E_1}, r_{E_1}, (\mathit{result}_{E_1}, \mathit{signal}_{E_1}), (\mathit{RegState}_{E_1}, \mathit{LabelState}_{E_1}, \mathit{EventState}_{E_1})) \; = \\ \mathbf{E}\;[E_1]\; l_{\mathit{trigger}_{E_1}}\;(\mathit{RegState}_{\mathit{result}}, \mathit{LabelState}_{\mathit{signal}}, \mathit{EventState}_{e_n})\; \mathit{env}_{E_1}$$

$$\vdots$$

$$(\mathbf{C}_{E_{n-1}}, r_{E_{n-1}}, (\mathit{result}_{E_{n-1}}, \mathit{signal}_{E_{n-1}}), \\ (\mathit{RegState}_{E_{n-1}}, \mathit{LabelState}_{E_{n-1}}, \mathit{EventState}_{E_{n-1}})) \; = \\ \mathbf{E}\;[E_{n-1}]\; l_{\mathit{split}_{E_{n-1}}}\;(\mathit{RegState}_{E_{n-2}}, \mathit{LabelState}_{E_{n-2}}, \mathit{EventState}_{E_{n-2}})\; \mathit{env}_{E_{n-1}}$$

$$(\mathbf{C}_{E_n}, r_{E_n}, (\mathit{result}_{E_n}, \mathit{signal}_{E_n}), (\mathit{RegState}_{E_n}, \mathit{LabelState}_{E_n}, \mathit{EventState}_{E_n})) \; = \\ \mathbf{E}\;[E_n]\; l_{\mathit{split}_{E_n}}\;(\mathit{RegState}_{E_{n-1}}, \mathit{LabelState}_{E_{n-1}}, \mathit{EventState}_{E_{n-1}})\; \mathit{env}'_{E_{n-1}}$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{\mathit{split}}, \mathit{LabelState}'', \mathit{EventState}'', l_{\mathit{split}}) \; = \\ \mathit{Split}([l_{\mathit{split}_{E_1}}, \ldots, l_{\mathit{split}_{E_n}}, l'_{\mathit{split}_{E_1}}, \ldots, l'_{\mathit{split}_{E_n}}]2n, \mathit{LabelState}_{l_n}, \mathit{EventState}_{E_n})$$

$$(\mathbf{C}_{ST}, \mathit{LabelState}', \mathit{EventState}', \mathit{signal}) \; = \\ \mathit{SignalTree}([\mathit{signal}_{E_1}, \ldots, \mathit{signal}_{E_n}, \mathit{end}_1, \ldots, \mathit{end}_n], 2n, \mathit{LabelState}'', \mathit{EventState}'')$$

— End of Splitting and Synchronizing —

Selections

$$\mathbf{E}\;[\;\texttt{selectN}\;(\texttt{numberN}\;n)\;E\;]\;\mathit{trigger}\;(\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\;\mathit{env} \; =$$

$$(\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'))$$

where

$$\mathbf{C} = \begin{pmatrix} & \mathbf{C}_E & \\ \mathit{result}_E: & \mathrm{READ} & n, r_E, +1 \\ & \mathrm{COPYI} & r, r, \mathit{result} \end{pmatrix}$$

$$\mathbf{IF}\; getFlag(E) \neq \textsf{pointer}\; \mathbf{THEN}\; \mathit{Error}\; \mathbf{END}$$

$$(r, RegStateR) \;=\; NewReg(RegState)$$

$$(e, EventStateE) \;=\; NewEvent(EventState)$$

$$result, signal, LabelStateSignal \;=\; NewLabel(LabelState_m)^2$$

$$\boxed{\begin{array}{l}(\mathbf{C}_E, r_E, (result_E, signal), (RegState', LabelState', EventState')) \;=\\ \mathbf{E}\;[E]\;trigger\;\;(RegStateR, LabelStateSignal, EventStateE)\;\;env\end{array}}$$

Advanced Constructs

Higher Order Functions (Function Call)

$$\begin{array}{l}\mathbf{E}\;[\;\texttt{applyN}\;E\;E_1\;\ldots\;E_n\;]\;trigger\;\;(RegState, LabelState, EventState)\;\;env\;\;=\\ (\mathbf{C}, r, (result, signal), (RegState_{E_n}, LabelState', EventState'))\end{array}$$

where

$$
\mathbf{C} \;=\; \left(
\begin{array}{lll}
 & \mathbf{C}_{split} & \\
 & & \\
 & \mathbf{C}_{0,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{0,m_0} & \\
 & & \\
 & \mathbf{C}_{E} & \\
 & & \\
result_E: & \mathrm{COPYP} & r_E, r_E, +1 \\
 & \mathrm{READ} & 0, r_E, +1 \\
 & \mathrm{APPLY} & r_1, +1 \\
 & \mathrm{SUBMIT} & r_E, r_{act}, send \\
 & \mathrm{COPYI} & r, r, result \\
 & & \\
 & \mathbf{C}_{send} & \\
 & & \\
 & \mathbf{C}_{1,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{1,m_1} & \\
 & & \\
 & \mathbf{C}_{E_1} & \\
 & & \\
result_{E_1}: & \mathrm{WAIT} & e_1, +2 \\
send_1: & \mathrm{WAIT} & e_1, +1 \\
 & \mathrm{SEND} & r_{E_1}, r_{act}, 1 \\
 & \mathrm{JUMP} & arg_1 \\
 & & \\
 & \quad\vdots & \\
 & & \\
 & \mathbf{C}_{n-1,1} & \\
 & \quad\vdots & \\
 & \mathbf{C}_{n-1,m_{n-1}} & \\
 & & \\
 & \mathbf{C}_{E_{n-1}} & \\
 & & \\
result_{E_{n-1}}: & \mathrm{WAIT} & e_{n-1}, +2 \\
send_{n-1}: & \mathrm{WAIT} & e_{n-1}, +1 \\
 & \mathrm{SEND} & r_{E_{n-1}}, r_{act}, n-1 \\
 & \mathrm{JUMP} & arg_{n-1} \\
 & & \\
 & \mathbf{C}_{E_n} & \\
 & & \\
result_{E_n}: & \mathrm{WAIT} & e_n, +2 \\
send_n: & \mathrm{WAIT} & e_n, +1 \\
 & \mathrm{SEND} & r_{E_n}, r_{act}, n \\
 & \mathrm{JUMP} & arg_n \\
 & & \\
 & \mathbf{C}_{ST} &
\end{array}
\right)
$$

$$\begin{aligned} fV &= freeVars(E) \\ fV_1 &= freeVars(E_1) \\ &\vdots \\ fV_n &= freeVars(E_n) \end{aligned}$$

$$r_1, r_{act}, r, RegState_{result} = NewReg(RegState)^3$$

$$arg_1, \ldots, arg_n, send, result, signal, LabelState_{signal} = NewLabel(LabelState)^{n+3}$$

$$e_1, \ldots, e_n, EventStateE = NewEvent(EventState)^n$$

— Forking common free variables —

$$\begin{aligned} &[y_{0,1}, \ldots, y_{0,m_0}] = fV \cap (fV_1 \cup \cdots \cup fV_n) \\ &(ry_{0,1}, ly_{0,1}, fy_{0,1}) = \\ &(lookupRegister\ env\ y_{0,1}\ ,\ lookupLabel\ env\ y_{0,1}\ ,\ lookupFlag\ env\ y_{0,1}) \\ &\qquad \vdots \\ &(ry_{0,m_0}, ly_{0,m_0}, fy_{0,m_0}) = \\ &(lookupRegister\ env\ y_{0,m_0}\ ,\ lookupLabel\ env\ y_{0,m_0}\ ,\ lookupFlag\ env\ y_{0,m_0}) \end{aligned}$$

$$\begin{aligned} &(\mathbf{C}_{0,1}, [l_{0,1}, l'_{0,1}], LabelState_{0,1}) = \\ &\overline{Split}(ly_{0,1}, ry_{0,1}, 2, fy_{0,1}, LabelState_{signal}) \\ &\qquad \vdots \\ &(\mathbf{C}_{0,m_0}, [l_{0,m_0}, l'_{0,m_0}], LabelState_{0,m_0}) = \\ &\overline{Split}(ly_{0,m_0}, ry_{0,m_0}, 2, fy_{0,m_0}, LabelState_{0,m_0-1}) \end{aligned}$$

$$\begin{aligned} env_E &= BindVar\ (y_{0,1}\ l_{0,1}) \circ \cdots \circ (y_{0,m_0}\ l_{0,m_0})\ env \\ env'_E &= BindVar\ (y_{0,1}\ l'_{0,1}) \circ \cdots \circ (y_{0,m_0}\ l'_{0,m_0})\ env \end{aligned}$$

$[y_{1,1}, \ldots, y_{1,m_1}] \; = \; fV_1 \; \cap (fV_2 \cup \cdots \cup fV_n)$
$(ry_{1,1}, ly_{1,1}, fy_{1,1}) \; =$
$(lookupRegister \; env \; y_{1,1} \; , \; lookupLabel \; env \; y_{1,1} \; ,$
$lookupFlag \; env \; y_{1,1})$

$$\vdots$$

$(ry_{1,m_1}, ly_{1,m_1}, fy_{1,m_1}) \; =$
$(lookupRegister \; env \; y_{1,m_1} \; , \; lookupLabel \; env \; y_{1,m_1} \; ,$
$lookupFlag \; env \; y_{1,m_1})$

$(\mathbf{C}_{1,1}, [l_{1,1}, l'_{1,1}], LabelState_{1,1}) \; = \overline{Split}(ly_{1,1}, ry_{1,1}, 2, fy_{1,1}, LabelState_{signal})$

$$\vdots$$

$(\mathbf{C}_{1,m_1}, [l_{1,m_1}, l'_{1,m_1}], LabelState_{1,m_1}) \; =$
$\overline{Split}(ly_{1,m_1}, ry_{1,m_1}, 2, fy_{1,m_1}, LabelState_{1,m_1-1})$

$env_{E_1} \; = \; BindVar \; (y_{1,1} \; l_{1,1}) \; \circ \cdots \circ (y_{1,m_1} \; l_{1,m_1}) \; env'_E$
$env'_{E_1} \; = \; BindVar \; (y_{1,1} \; l'_{1,1}) \; \circ \cdots \circ (y_{1,m_1} \; l'_{1,m_1}) \; env'_E$

$$\vdots$$

$[y_{n-1,1}, \ldots, y_{n-1,m_{n-1}}] \; = \; fV_{n-1} \; \cap fV_n$
$(ry_{n-1,1}, ly_{n-1,1}, fy_{n-1,1}) \; =$
$(lookupRegister \; env'_{E_{n-2}} \; y_{n-1,1} \; , \; lookupLabel \; env'_{E_{n-2}} \; y_{n-1,1} \; ,$
$lookupFlag \; env'_{E_{n-2}} \; y_{n-1,1})$

$$\vdots$$

$(ry_{n-1,m_{n-1}}, ly_{n-1,m_{n-1}}, fy_{n-1,m_{n-1}}) \; =$
$(lookupRegister \; env'_{E_{n-2}} \; y_{n-1,m_{n-1}} \; , \; lookupLabel \; env'_{E_{n-2}} \; y_{n-1,m_{n-1}} \; ,$
$lookupFlag \; env'_{E_{n-2}} \; y_{n-1,m_{n-1}})$

$(\mathbf{C}_{n-1,1}, [l_{n-1,1}, l'_{n-1,1}], LabelState_{n-1,1}) \; =$
$\overline{Split}(ly_{n-1,1}, ry_{n-1,1}, 2, fy_{n-1,1}, LabelState_{n-2,m_{n-2}})$

$$\vdots$$

$(\mathbf{C}_{n-1,m_{n-1}}, [l_{n-1,m_{n-1}}, l'_{n-1,m_{n-1}}], LabelState_{n-1,m_{n-1}}) \; =$
$\overline{Split}(ly_{n-1,m_{n-1}}, ry_{n-1,m_{n-1}}, 2, fy_{n-1,m_{n-1}}, LabelState_{n-1,m_{n-1}-1})$

$env_{E_{n-1}} \; = \; BindVar \; (y_{n-1,1} \; l_{n-1,1}) \; \circ \cdots \circ (y_{n-1,m_{n-1}} \; l_{n-1,m_{n-1}}) \; env'_{E_{n-2}}$
$env'_{E_{n-1}} \; = \; BindVar \; (y_{n-1,1} \; l'_{n-1,1}) \; \circ \cdots \circ (y_{n-1,m_{n-1}} \; l'_{n-1,m_{n-1}}) \; env'_{E_{n-2}}$

— End of forking common free variables —

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState_E, LabelState_E, EventState_E)) =$$
$$\mathbf{E}\ [E]\ l_{split_0}\ (RegState_{result}, LabelState_{signal}, EventStateE)\ env_E$$

$$(\mathbf{C}_{E_1}, r_{E_1}, (result_{E_1}, signal_{E_1}), (RegState_{E_1}, LabelState_{E_1}, EventState_{E_1})) =$$
$$\mathbf{E}\ [E_1]\ l_{split_1}\ (RegState_{result}, LabelState_{signal}, EventState_{e_n})\ env_{E_1}$$

$$\vdots$$

$$(\mathbf{C}_{E_{n-1}}, r_{E_{n-1}}, (result_{E_{n-1}}, signal_{E_{n-1}}),$$
$$(RegState_{E_{n-1}}, LabelState_{E_{n-1}}, EventState_{E_{n-1}})) =$$
$$\mathbf{E}\ [E_{n-1}]\ l_{split_{n-1}}\ (RegState_{E_{n-2}}, LabelState_{E_{n-2}}, EventState_{E_{n-2}})\ env_{E_{n-1}}$$

$$(\mathbf{C}_{E_n}, r_{E_n}, (result_{E_n}, signal_{E_n}), (RegState_{E_n}, LabelState_{E_n}, EventState_{E_n})) =$$
$$\mathbf{E}\ [E_n]\ l_{split_n}\ (RegState_{E_{n-1}}, LabelState_{E_{n-1}}, EventState_{E_{n-1}})\ env'_{E_{n-1}}$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{split}, [l_{split_0}, \ldots, l_{split_n}], LabelState_{split}) = Split(trigger, n+1, LabelState_{E_n})$$

$$(\mathbf{C}_{send}, [send_1, \ldots, send_n], LabelState_{send}) = SplitP(send, r_{act}, n, LabelState_{split})$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) =$$
$$SignalTree([arg_1, \ldots, arg_n, signal_E, signal_{E_1}, \ldots, signal_{E_n}],$$
$$2n+1, LabelState_{send}, EventState_{E_n})$$

— End of Splitting and Synchronizing —

Curried Expressions

$$\mathbf{E}\ [\ \texttt{curryN}\ (\texttt{idN}\ f)\ (\texttt{idN}\ v_1)\ \ldots\ (\texttt{idN}\ v_n)\ ]\ trigger$$
$$(RegState, LabelState, EventState)\ env =$$
$$(\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState'))$$

where

$$
\mathbf{C} = \left(\begin{array}{lll}
\mathit{trigger}: & \text{ALLOC} & n+1, +1 \\
 & \text{COPYP} & r, r, +1 \\
 & \text{SPLIT} & \mathit{result}, \mathit{split}' \\
 & & \\
 & \mathbf{C}_{splitP} & \\
 & & \\
l_{split_0}: & \text{CONST} & r_0, +2 \\
 & < f, i > & \\
 & \text{WRITEI} & r_0, r, 0 \\
 & \text{JUMP} & \mathit{end} \\
 & & \\
l_{split_1}: & \text{COPYP} & r, r, \mathit{trigger}_1 \\
 & \mathbf{C}_{v_1} & \\
 & \mathbf{C}'_{v_1} & \\
 & \text{JUMP} & \mathit{end}_1 \\
 & & \\
 & \vdots & \\
 & & \\
l_{split_{n-1}}: & \text{COPYP} & r, r, \mathit{trigger}_{n-1} \\
 & \mathbf{C}_{v_{n-1}} & \\
 & \mathbf{C}'_{v_{n-1}} & \\
 & \text{JUMP} & \mathit{end}_{n-1} \\
 & & \\
 & \mathbf{C}_{v_n} & \\
 & \mathbf{C}'_{v_n} & \\
 & \text{JUMP} & \mathit{end}_n \\
 & & \\
 & \mathbf{C}_{ST} &
\end{array} \right)
$$

$$
\mathbf{C}'_{v_1} = \begin{cases} \left(\mathit{result}_1: \ \text{WRITEI} \ \ r_1, r, 1 \right) & \textbf{if } \mathit{flag}_1 = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}_1: & \text{COPYP} & r_1, r_1, +1 \\ & \text{WRITEP} & r_1, r, 1 \end{array} \right) & \textbf{otherwise} \end{cases}
$$

$$\vdots$$

$$
\mathbf{C}'_{v_{n-1}} = \begin{cases} \left(\mathit{result}_{n-1}: \ \text{WRITEI} \ \ r_{n-1}, r, n-1 \right) & \textbf{if } \mathit{flag}_{n-1} = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}_{n-1}: & \text{COPYP} & r_{n-1}, r_{n-1}, +1 \\ & \text{WRITEP} & r_{n-1}, r, n-1 \end{array} \right) & \textbf{otherwise} \end{cases}
$$

$$
\mathbf{C}'_{v_n} = \begin{cases} \left(\mathit{result}_n: \ \text{WRITEI} \ \ r_n, r, n \right) & \textbf{if } \mathit{flag}_n = \mathsf{int} \\ \left(\begin{array}{lll} \mathit{result}_n: & \text{COPYP} & r_n, r_n, +1 \\ & \text{WRITEP} & r_n, r, n \end{array} \right) & \textbf{otherwise} \end{cases}
$$

$$\begin{aligned} \mathit{flag}_1 &= \mathit{flag}(v_1) \\ &\vdots \\ \mathit{flag}_n &= \mathit{flag}(v_n) \end{aligned}$$

$$r, r_0, \mathit{RegState}_r = \mathit{NewReg}(\mathit{RegState})^2$$

$$\begin{aligned} &\mathit{split}', \mathit{result}, \mathit{signal}, \mathit{trigger}_1, \ldots, \mathit{trigger}_{n-1}, \\ &\mathit{end}, \mathit{end}_1, \ldots, \mathit{end}_n, \mathit{LabelState}_{\mathit{signal}} = \\ &\mathit{NewLabel}(\mathit{LabelState})^{2n+3} \end{aligned}$$

$$i = \mathit{arity}(f) - (n-1)$$

$$\boxed{\begin{aligned} &(\mathbf{C}_{v_1}, r_{v_1}, (\mathit{result}_{v_1}, \mathit{signal}_{v_1}), (\mathit{RegState}_{v_1}, \mathit{LabelState}_{v_1}, \mathit{EventState}_{v_1})) = \\ &\mathbf{E}\,[v_1]\ \mathit{trigger}_1\ (\mathit{RegState}_E, \mathit{LabelState}_E, \mathit{EventState}_E)\ \mathit{env} \\ &\qquad\vdots \\ &(\mathbf{C}_{v_{n-1}}, r_{v_{n-1}}, (\mathit{result}_{v_{n-1}}, \mathit{signal}_{v_{n-1}}), \\ &(\mathit{RegState}_{v_{n-1}}, \mathit{LabelState}_{v_{n-1}}, \mathit{EventState}_{v_{n-1}})) = \\ &\mathbf{E}\,[v_{n-1}]\ \mathit{trigger}_{n-1}\ (\mathit{RegState}_{v_{n-2}}, \mathit{LabelState}_{v_{n-2}}, \mathit{EventState}_{v_{n-2}})\ \mathit{env} \\ &(\mathbf{C}_{v_n}, r_{v_n}, (\mathit{result}_{v_n}, \mathit{signal}_{v_n}), (\mathit{RegState}_{v_n}, \mathit{LabelState}_{v_n}, \mathit{EventState}_{v_n})) = \\ &\mathbf{E}\,[v_n]\ l_{\mathit{split}_n}\ (\mathit{RegState}_{v_{n-1}}, \mathit{LabelState}_{v_{n-1}}, \mathit{EventState}_{v_{n-1}})\ \mathit{env} \end{aligned}}$$

— Splitting and Synchronizing —

$$\begin{aligned} &(\mathbf{C}_{\mathit{splitP}}, [l_{\mathit{split}_0}, l_{\mathit{split}_1}, \ldots, l_{\mathit{split}_n}], \mathit{LabelState}_{\mathit{trigger}_0}) = \\ &\mathit{SplitP}(\mathit{split}', r, n+1, \mathit{LabelState}_{v_n}) \end{aligned}$$

$$\begin{aligned} &(\mathbf{C}_{ST}, \mathit{LabelState}', \mathit{EventState}', \mathit{signal}) = \\ &\mathit{SignalTree}([\mathit{signal}_{v_1}, \ldots, \mathit{signal}_{v_n}, \mathit{end}_1, \ldots, \mathit{end}_n, \mathit{end}], \\ &\qquad 2n+1, \mathit{LabelState}_{\mathit{split}_n}, \mathit{EventState}_{v_n}) \end{aligned}$$

— End of Splitting and Synchronizing —

B.3.2 Valuation Function of *Definition*

$$\mathbf{D} : \mathit{Definition} \rightarrow \mathit{Label} \rightarrow \mathit{State} \rightarrow [\mathit{Variable}] \rightarrow \mathit{Environment} \rightarrow \mathit{DefResult}$$

Simple Definitions

$$\begin{array}{l} \mathbf{D}\ [\ \texttt{defN}\ (\texttt{idN}\ x)\ E\ ]\ \mathit{trigger} \\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{vl}\ \mathit{env}\ = \\ (\mathbf{C}, \mathit{signal}, (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'), \mathit{env}') \end{array}$$

where

$$\mathbf{C} = \begin{pmatrix} & \mathbf{C}_E & \\ \mathit{result}_E : & \text{JUMP} & \mathit{result} \end{pmatrix}$$

$$f = \mathit{flag}(E)$$

$$\mathit{result}, \mathit{signal}, \mathit{LabelState}_{\mathit{signal}} = \mathit{NewLabel}(\mathit{LabelState})^2$$

$$\mathit{env}' = \mathit{env} \cup \{x \rightarrow (r_E, \mathit{result}, f)\}$$

$$\boxed{\begin{array}{l} (\mathbf{C}_E, r_E, (\mathit{result}_E, \mathit{signal}), (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}')) = \\ \mathbf{E}\ [E]\ \mathit{trigger}\ (\mathit{RegState}_r, \mathit{LabelState}_{\mathit{signal}}, \mathit{EventState})\ \mathit{env}' \end{array}}$$

Function Definitions

$$\begin{array}{l} \mathbf{D}\ [\ \texttt{defN}\ (\texttt{idN}\ x)\ (\texttt{lambdaN}\ (\texttt{lheadN}\ (\texttt{idN}\ x_1)\ \ldots\ (\texttt{idN}\ x_n)\ )\ (\texttt{lbodyN}\ E))\ ]\ \mathit{trigger} \\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{vl}\ \mathit{env}\ = \\ (\mathbf{C}, \mathit{signal}, (\mathit{RegState}', \mathit{LabelStateE}, \mathit{EventStateE}), \mathit{env}') \end{array}$$

where

$$\mathbf{C} = \begin{pmatrix} \mathit{trigger} : & \text{JUMP} & \mathit{start} \\ & \text{COPYI} & r_1, r_1, l_1 \\ & \vdots & \\ & \text{COPYI} & r_n, r_n, l_n \\ & \mathbf{C}_E & \\ \mathit{result}_E : & \text{RETURN} & r, +1 \\ \mathit{signal}_E : & \text{WAIT} & e, +1 \\ & \text{TERMINATE} & \end{pmatrix}$$

$$\begin{aligned} f_1 &= \mathit{flag}(x_1) \\ &\vdots \\ f_n &= \mathit{flag}(x_n) \\ f &= \mathit{flag}(E) \end{aligned}$$

$$\begin{aligned} &start, l_1, \ldots, l_n, result, LabelStateLn = NewLabel(LabelState)^{n+2} \\ &r, r_1, \ldots, r_n, RegStateRn = NewReg(RegState)^{n+1} \\ &e, EventStateE = NewEvent(EventState) \end{aligned}$$

$$\begin{aligned} envEin &= env \cup \{x_1 \rightarrow (r_1, l_1, f_1), \ldots, x_n \rightarrow (r_n, l_n, f_n)\} \\ &\quad \cup \{x \rightarrow (r, result, f)\} \\ env' &= env \cup \{x \rightarrow (r, result, f)\} \\ signal &= EmptyLabel \end{aligned}$$

$$\boxed{\begin{aligned} &(\mathbf{C}_E, r, (result_E, signal_E), (RegState_E, LabelState_E, EventState_E)) = \\ &\mathbf{E}\ [E]\ start\ (RegStateRn, LabelStateLn, EventStateE)\ envEin \end{aligned}}$$

Definition Sequences

$$\begin{aligned} &\mathbf{D}\ [\ \texttt{defseqN}\ D_1, \ldots, D_n\ ]\ trigger\ (RegState, LabelState, EventState)\ fV\ env\ = \\ &(\mathbf{C}, signal, (RegState', LabelState', EventState'), env_n) \end{aligned}$$

where

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_{split} \\ \\ \mathbf{C}_{1,1} \\ \vdots \\ \mathbf{C}_{1,m_1} \\ \mathbf{C}_{D_1} \\ \\ \vdots \\ \\ \mathbf{C}_{n-1,1} \\ \vdots \\ \mathbf{C}_{n-1,m_{n-1}} \\ \mathbf{C}_{D_{n-1}} \\ \mathbf{C}_{D_n} \\ \mathbf{C}_{ST} \end{pmatrix}$$

$$\begin{aligned} fV_1 &= freeVars(D_1) \\ &\vdots \\ fV_n &= freeVars(D_n) \end{aligned}$$

— Forking common free variables —

$[y_{1,1}, \ldots, y_{1,m_1}] \ = \ fV_1 \ \cap (fV_2 \cup \cdots \cup fV_n \cup fV)$
$(ry_{1,1}, ly_{1,1}, fy_{1,1}) \ =$
$(\mathit{lookupRegister} \ env \ y_{1,1} \ , \ \mathit{lookupLabel} \ env \ y_{1,1} \ ,$
$\mathit{lookupFlag} \ env \ y_{1,1})$

$$\vdots$$

$(ry_{1,m_1}, ly_{1,m_1}, fy_{1,m_1}) \ =$
$(\mathit{lookupRegister} \ env \ y_{1,m_1} \ , \ \mathit{lookupLabel} \ env \ y_{1,m_1} \ ,$
$\mathit{lookupFlag} \ env \ y_{1,m_1})$

$\underline{(\mathbf{C}_{1,1}, [l_{1,1}, l'_{1,1}], \mathit{LabelState}_{1,1}) \ =}$
$\overline{\mathit{Split}}(ly_{1,1}, ry_{1,1}, 2, fy_{1,1}, \mathit{LabelState}_{signal})$

$$\vdots$$

$\underline{(\mathbf{C}_{1,m_1}, [l_{1,m_1}, l'_{1,m_1}], \mathit{LabelState}_{1,m_1}) \ =}$
$\overline{\mathit{Split}}(ly_{1,m_1}, ry_{1,m_1}, 2, fy_{1,m_1}, \mathit{LabelState}_{1,m_1-1})$

$env_{D_1} \ = \ \mathit{BindVar} \ (y_{1,1} \ l_{1,1}) \ \circ \cdots \circ (y_{1,m_1} \ l_{1,m_1}) \ env$

$env''_{D_1} \ = \ \mathit{BindVar} \ (y_{1,1} \ l'_{1,1}) \ \circ \cdots \circ (y_{1,m_1} \ l'_{1,m_1}) \ env'_{D_1}$

$$\vdots$$

$[y_{n-1,1}, \ldots, y_{n-1,m_{n-1}}] \ = \ fV_{n-1} \ \cap \ (fV_n \ \cup \ fV)$
$(ry_{n-1,1}, ly_{n-1,1}, fy_{n-1,1}) \ =$
$(\mathit{lookupRegister} \ env'_{E_{n-2}} \ y_{n-1,1} \ , \ \mathit{lookupLabel} \ env'_{E_{n-2}} \ y_{n-1,1} \ ,$
$\mathit{lookupFlag} \ env'_{E_{n-2}} \ y_{n-1,1})$

$$\vdots$$

$(ry_{n-1,m_{n-1}}, ly_{n-1,m_{n-1}}, fy_{n-1,m_{n-1}}) \ =$
$(\mathit{lookupRegister} \ env'_{E_{n-2}} \ y_{n-1,m_{n-1}} \ , \ \mathit{lookupLabel} \ env'_{E_{n-2}} \ y_{n-1,m_{n-1}} \ ,$
$\mathit{lookupFlag} \ env'_{E_{n-2}} \ y_{n-1,m_{n-1}})$

$\underline{(\mathbf{C}_{n-1,1}, [l_{n-1,1}, l'_{n-1,1}], \mathit{LabelState}_{n-1,1}) \ =}$
$\overline{\mathit{Split}}(ly_{n-1,1}, ry_{n-1,1}, 2, fy_{n-1,1}, \mathit{LabelState}_{n-2,m_{n-2}})$

$$\vdots$$

$\underline{(\mathbf{C}_{n-1,m_{n-1}}, [l_{n-1,m_{n-1}}, l'_{n-1,m_{n-1}}], \mathit{LabelState}_{n-1,m_{n-1}}) \ =}$
$\overline{\mathit{Split}}(ly_{n-1,m_{n-1}}, ry_{n-1,m_{n-1}}, 2, fy_{n-1,m_{n-1}}, \mathit{LabelState}_{n-1,m_{n-1}-1})$

$env_{D_{n-1}} \ = \ \mathit{BindVar} \ (y_{n-1,1} \ l_{n-1,1}) \ \circ \cdots \circ (y_{n-1,m_{n-1}} \ l_{n-1,m_{n-1}}) \ env''_{D_{n-2}}$

$env''_{D_{n-1}} \ = \ \mathit{BindVar} \ (y_{n-1,1} \ l'_{n-1,1}) \ \circ \cdots \circ (y_{n-1,m_{n-1}} \ l'_{n-1,m_{n-1}}) \ env'_{D_{n-1}}$

$$[y_{n,1},\ldots,y_{n,m_n}] \;=\; fV_n \;\cap\; fV$$
$$(ry_{n,1}, ly_{n,1}, fy_{n,1}) \;=$$
$$(lookupRegister\ env'_{E_{n-2}}\ y_{n,1}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n,1}\ ,$$
$$lookupFlag\ env'_{E_{n-2}}\ y_{n,1})$$
$$\vdots$$
$$(ry_{n,m_n}, ly_{n,m_n}, fy_{n,m_n}) \;=$$
$$(lookupRegister\ env'_{E_{n-2}}\ y_{n,m_n}\ ,\ lookupLabel\ env'_{E_{n-2}}\ y_{n,m_n}\ ,$$
$$lookupFlag\ env'_{E_{n-2}}\ y_{n,m_n})$$

$$\underline{(\mathbf{C}_{n,1}, [l_{n,1}, l'_{n,1}], LabelState_{n,1})} \;=$$
$$\overline{Split}(ly_{n,1}, ry_{n,1}, 2, fy_{n,1}, LabelState_{n-2,m_{n-2}})$$
$$\vdots$$
$$\underline{(\mathbf{C}_{n,m_n}, [l_{n,m_n}, l'_{n,m_n}], LabelState_{n,m_n})} \;=$$
$$\overline{Split}(ly_{n,m_n}, ry_{n,m_n}, 2, fy_{n,m_n}, LabelState_{n,m_n-1})$$

$$env_{D_n} \;=\; BindVar\ (y_{n,1}\ l_{n,1})\ \circ\cdots\circ (y_{n,m_n}\ l_{n,m_n})\ env''_{D_{n-2}}$$

— End of forking common free variables —

$$(\mathbf{C}_{D_1}, signal_1, (RegState_1, LabelState_1, EventState_1), env'_{D_1}) \;=$$
$$\mathbf{D}\ [D_1]\ l_{split_1}\ (RegState_{result}, LabelState_{result}, EventState)\ env_{D_1}$$

$$\vdots$$

$$(\mathbf{C}_{D_{n-1}}, signal_{n-1}, (RegState_{n-1}, LabelState_{n-1}, EventState_{n-1}), env'_{D_{n-1}}) \;=$$
$$\mathbf{D}\ [D_{n-1}]\ l_{split_{n-1}}\ (RegState_{n-2}, LabelState_{n-2}, EventState_{n-2})\ env_{D_{n-1}}$$

$$(\mathbf{C}_{D_n}, signal, (RegState', LabelState_n, EventState_n), env_n) \;=$$
$$\mathbf{D}\ [D_n]\ l_{split_n}\ (RegState_{n-1}, LabelState_{n-1}, EventState_{n-1})\ env_{D_n}$$

— Splitting and Synchronizing —

$$(\mathbf{C}_{split}, [l_{split_1},\ldots,l_{split_n}], LabelState') \;=\; Split(trigger, n, LabelState_n)$$

$$\begin{array}{l}
\mathbf{IF}\, signal_1 \neq EmptyLabel \\
\quad \mathbf{THEN}\ (\mathbf{C}_{ST}, LabelState', EventState', signal) \;= \\
\qquad\qquad SignalTree([signal_1,\ldots,signal_n], n, LabelState_n, EventState_n) \\
\quad \mathbf{ELSE}\ (\mathbf{C}_{ST}, LabelState', EventState', signal) \;= \\
\qquad\qquad ((), LabelState_n, EventState_n, EmptyLabel) \\
\mathbf{END}
\end{array}$$

— End of Splitting and Synchronizing —

B.3.3 Valuation Function of *Program*

$$\mathbf{P} \ : \ \mathit{Program} \rightarrow \mathit{Result}$$

$$\mathbf{P} \, [\, \texttt{programN} \ DS \,] \ = \ \mathbf{C}$$

where

$$(\mathbf{C}, \mathit{signal}', (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'), \mathit{env}') \ = \\ \mathbf{D} \, [DS] \ \mathit{StartLabel} \ (\mathit{InitReg}, \mathit{InitLabel}, \mathit{InitEvent}) \ \mathit{InitEnv}$$

B.4 Inline Code for Predefined Functions

Absolute Value

$$\mathbf{E} \, [\, \texttt{callN (idN ABS)} \ E \,] \ \mathit{trigger} \ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState}) \ \mathit{env} \ = \\ (\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'))$$

where

$$\mathbf{C} \ = \ \left(\begin{array}{lll} & \mathbf{C}_E & \\ \mathit{result}_E : & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{LESSC} & 0, r_E, +1 \\ & \mathrm{SWITCH} & r_E, l_{\mathit{true}}, l_{\mathit{false}} \\ & \mathrm{COPYI} & r, r_E, l_{\mathit{false}} \\ l_{\mathit{true}} : & \mathrm{NEG} & r, +1 \\ l_{\mathit{false}} : & \mathrm{SPLIT} & \mathit{result}, \mathit{end} \\ & \mathbf{C}_{ST} & \end{array} \right)$$

$$r, \mathit{RegStateR} \ = \ \mathit{NewReg}(\mathit{RegState})$$

$$\mathit{result}, \mathit{signal}, \mathit{end}, l_{\mathit{true}}, l_{\mathit{false}}, \mathit{LabelStateL} \ = \ \mathit{NewLabel}(\mathit{LabelState})^5$$

$$(\mathbf{C}_E, r_E, (\mathit{result}_E, \mathit{signal}_E), (\mathit{RegState}', \mathit{LabelState}_E, \mathit{EventState}_E)) \ = \\ \mathbf{E} \, [E] \ \mathit{trigger} \ (\mathit{RegStateR}, \mathit{LabelStateL}, \mathit{EventState}) \ \mathit{env}$$

$$(\mathbf{C}_{ST}, \mathit{LabelState}', \mathit{EventState}', \mathit{signal}) \ = \\ \mathit{SignalTree}([\mathit{signal}_E, \mathit{end}], 2, \mathit{LabelState}_E, \mathit{EventState}_E)$$

Test Whether a Number Is Odd

$$\mathbf{E}\ [\ \texttt{callN (idN ODD)}\ E\ ]\ \mathit{trigger}\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{env}\ =\ (\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'))$$

where

$$\mathbf{C}\ =\ \left(\begin{array}{lll} & \mathbf{C}_E & \\ \mathit{result}_E: & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{MODC} & 2, r, +1 \\ & \mathrm{EQUALC} & 0, r, +1 \\ & \mathrm{NOT} & r, +1 \\ & \mathrm{SPLIT} & \mathit{result}, \mathit{end} \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$r, \mathit{RegStateR}\ =\ \mathit{NewReg}(\mathit{RegState})$$

$$\mathit{result}, \mathit{signal}, \mathit{end}, \mathit{LabelStateL}\ =\ \mathit{NewLabel}(\mathit{LabelState})^3$$

$$(\mathbf{C}_E, r_E, (\mathit{result}_E, \mathit{signal}_E), (\mathit{RegState}', \mathit{LabelState}_E, \mathit{EventState}_E))\ =\ \mathbf{E}\ [E]\ \mathit{trigger}\ (\mathit{RegStateR}, \mathit{LabelStateL}, \mathit{EventState})\ \mathit{env}$$

$$(\mathbf{C}_{ST}, \mathit{LabelState}', \mathit{EventState}', \mathit{signal})\ =\ \mathit{SignalTree}([\mathit{signal}_E, \mathit{end}], 2, \mathit{LabelState}_E, \mathit{EventState}_E)$$

Extract Head of a List

$$\mathbf{E}\ [\ \texttt{callN (idN HEAD)}\ E\ ]\ \mathit{trigger}\ (\mathit{RegState}, \mathit{LabelState}, \mathit{EventState})\ \mathit{env}\ =\ (\mathbf{C}, r, (\mathit{result}, \mathit{signal}), (\mathit{RegState}', \mathit{LabelState}', \mathit{EventState}'))$$

where

$$\mathbf{C}\ =\ \left(\begin{array}{lll} & \mathbf{C}_E & \\ \mathit{result}_E: & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{READ} & 0, r, +1 \\ & \mathrm{COPYI} & r, r, +1 \\ & \mathrm{SPLIT} & \mathit{result}, \mathit{end} \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$r, \mathit{RegStateR}\ =\ \mathit{NewReg}(\mathit{RegState})$$

$$result, signal, end, LabelStateL \;=\; NewLabel(LabelState)^3$$

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState', LabelState_E, EventState_E)) \;=\; \\ \mathbf{E}\,[E]\; trigger\; (RegStateR, LabelStateL, EventState)\; env$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \;=\; \\ SignalTree([signal_E, end], 2, LabelState_E, EventState_E)$$

Extract Tail of a List

$$\mathbf{E}\,[\; \texttt{callN (idN TAIL)}\; E\;]\; trigger\; (RegState, LabelState, EventState)\; env \;=\; \\ (\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState'))$$

where

$$\mathbf{C} \;=\; \left(\begin{array}{lll} & \mathbf{C}_E & \\ result_E : & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{READ} & 1, r, +1 \\ & \mathrm{COPYI} & r, r, +1 \\ & \mathrm{SPLIT} & result, end \\ & \mathbf{C}_{ST} & \end{array} \right)$$

$$r, RegStateR \;=\; NewReg(RegState)$$

$$result, signal, end, LabelStateL \;=\; NewLabel(LabelState)^3$$

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState', LabelState_E, EventState_E)) \;=\; \\ \mathbf{E}\,[E]\; trigger\; (RegStateR, LabelStateL, EventState)\; env$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \;=\; \\ SignalTree([signal_E, end], 2, LabelState_E, EventState_E)$$

Capitalize Letter

$$\mathbf{E}\ [\ \texttt{callN (idN CAP)}\ E\ ]\ trigger\ (RegState, LabelState, EventState)\ env\ =$$
$$(\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState'))$$

where

$$\mathbf{C}\ =\ \left(\begin{array}{lll} & \mathbf{C}_E & \\ result_E: & \text{COPYI} & r, r_E, +1 \\ & \text{COPYI} & r_1, r_E, +1 \\ & \text{LESSRC} & 64, r_1, +1 \\ & \text{COPYI} & r_2, r_E, +1 \\ & \text{LESSC} & 123, r_2, +1 \\ & \text{AND} & r_1, r_2, +1 \\ & \text{SWITCH} & r_1, l_{true}, l_{false} \\ & \text{COPYI} & r, r_1, l_{false} \\ l_{true}: & \text{SUBC} & 32, r, +1 \\ l_{false}: & \text{SPLIT} & result, end \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$r, r_1, r_2, RegStateR\ =\ NewReg(RegState)^3$$

$$result, signal, end, l_{true}, l_{false}, LabelStateL\ =\ NewLabel(LabelState)^5$$

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState', LabelState_E, EventState_E))\ =$$
$$\mathbf{E}\ [E]\ trigger\ (RegStateR, LabelStateL, EventState)\ env$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal)\ =$$
$$SignalTree([signal_E, end], 2, LabelState_E, EventState_E)$$

Test Whether List is Empty

$$\mathbf{E}\ [\ \texttt{callN (idN NULL)}\ E\ ]\ trigger\ (RegState, LabelState, EventState)\ env\ =$$
$$(\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState'))$$

where

$$\mathbf{C} \;=\; \left(\begin{array}{lll} & \mathbf{C}_E & \\ result_E: & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{EQUALC} & 0, r, +1 \\ & \mathrm{SPLIT} & result, end \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$r, RegStateR \;=\; NewReg(RegState)$$

$$result, signal, end, LabelStateL \;=\; NewLabel(LabelState)^3$$

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState', LabelState_E, EventState_E)) \;=\;$$
$$\mathbf{E}\;[E]\; trigger\; (RegStateR, LabelStateL, EventState)\; env$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \;=\;$$
$$SignalTree([signal_E, end], 2, LabelState_E, EventState_E)$$

Test Whether Some Computation Has Failed

$$\mathbf{E}\;[\;\texttt{callN (idN FAILED)}\; E\;]\; trigger\; (RegState, LabelState, EventState)\; env \;=\;$$
$$(\mathbf{C}, r, (result, signal), (RegState', LabelState', EventState'))$$

where

$$\mathbf{C} \;=\; \left(\begin{array}{lll} & \mathbf{C}_E & \\ result_E: & \mathrm{COPYI} & r, r_E, +1 \\ & \mathrm{FAILED} & r, +1 \\ & \mathrm{SPLIT} & result, end \\ & \mathbf{C}_{ST} & \end{array}\right)$$

$$r, RegStateR \;=\; NewReg(RegState)$$

$$result, signal, end, LabelStateL \;=\; NewLabel(LabelState)^3$$

$$(\mathbf{C}_E, r_E, (result_E, signal_E), (RegState', LabelState_E, EventState_E)) \;=\;$$
$$\mathbf{E}\;[E]\; trigger\; (RegStateR, LabelStateL, EventState)\; env$$

$$(\mathbf{C}_{ST}, LabelState', EventState', signal) \;=\;$$
$$SignalTree([signal_E, end], 2, LabelState_E, EventState_E)$$

Appendix C

Description of the ADAM Instruction Set

This appendix describes the instruction set of the ADAM machine which has itself been described in Section 3. This description is split into two parts:

1. First the **syntax** of ADAM assembler code and of the files that are created by the EVE compiler are described and then

2. an informal description of the **semantics** of the ADAM instructions is given.

However, the instruction set will only be discussed in as much detail as it is necessary to understand the code that is generated by the EVE compiler. For a more detailed discussion of the ADAM instruction set than it is given in this appendix and its implementation see [Schreiner, 1990][Appendix B].

Figure C.1 shows a summary of the ADAM instruction set[1]. This picture is taken from the above mentioned diploma thesis.

C.1 Syntax

In the syntax description, the distinction of ADAM objects into registers, labels, events and functions was already given by using different non-terminal symbols for their description. However, in this part it will only be described which kind of object may appear at which position of an instruction. The operations have been grouped to several classes according to their type i.e. according to the kinds of arguments they need. This grouping of all ADAM instructions is described in Section C.1. Based on this classification, in Section C.1 the general structure of one line of ADAM assembler code is described. In general, ADAM assembler code is then just a sequence of such lines (see Section C.1).

After the general description of ADAM assembler code, in the next sections the *structure of the output files* that are generated by the compiler are described. These files are:

[1]Note that the names of the types in Figure C.1 correspond to the names of the non-terminal symbols of the syntax description in Section C.1.

Name	**Type**	**Description**
ADD(C)	OpReReIn	Integer Addition (with Constant)
SUB((R)C)	OpReReIn	Integer Subtraction ((Reverse) with Constant)
MUL(C)	OpReReIn	Integer Multiplication (with Constant)
DIV((R)C)	OpReReIn	Integer Division ((Reverse) with Constant)
MOD((R)C)	OpReReIn	Integer Remainder ((Reverse) with Constant)
AND	OpReReIn	Boolean Conjunction
OR	OpReReIn	Boolean Disjunction
EQUAL(C)	OpReReIn	Test for Integer Equality (with Constant)
LESS((R)C)	OpReReIn	Test for Integer Strict Order ((Reverse) with Constant)
LESSEQ((R)C)	OpReReIn	Test for Integer Reflexive Order ((Reverse) with Constant)
NEG	OpReReIn	Integer Negation
NOT	OpReReIn	Boolean Negation
FAILED	OpReReIn	Test for Error Value
COPYI	OpReReIn	Duplicate an Integer Value
COPYP	OpReReIn	Duplicate a Tuple Pointer
DELP	OpReIn	Delete a Pointer Value
SPLIT	OpInIn	Split Control Flow Into Parallel Paths
SWITCH	OpReInIn	Switch Control Flow
JUMP	OpJu	Route Control Flow
WAIT	OpEvIn	Wait for Synchronization Event
CONST	OpReIn	Generate Long Constant
FASTCALL	OpFuIn	Request Standard Function Activation
TAILCALL	OpFuIn	Request Tail-Recursive Function Activation
CONTEXT	OpReIn	Submit Current Result Address to Callee
APPLY	OpReIn	Request Activation of a Non-Fixed Function
SEND(C)	OpReReCo	Send (Constant) Argument to a Function Activation
SUBMIT	OpReReIn	Request Submission of Curried Function Arguments
RETURN(C)	OpReIn	Send Result Value back to Caller
TERMINATE	Op	Terminate Actual Activation
ALLOC	OpCoIn	Allocate Tuple
READ	OpCoReIn	Read Contents of a Tuple Field
WRITEI	OpReReCo	Write Integer into a Tuple Field
WRITEP	OpReReCo	Write Pointer into a Tuple Field
WRITEC	OpCoReCo	Write Integer Constant into a Tuple Field

Figure C.1: The ADAM Instruction Set (Summary)

- A *program file* (with extension "*.asm*"), which contains for each function of the EVE program the corresponding ADAM assembler code. However, in this file the code for a function has to be embedded in other statements containing the name of the function and the necessary number of registers and events. The whole program file must also contain a list of those functions that can be called from outside this file ('export' list). The structure of this file is described in Section C.1.
- An *expression file* (with extension "*.ase*"), which contains the code that is generated from the body of the definition of the EVE constant `MAIN`. This code is embedded in statements that announce the ADAM functions that may be called from inside this expression. The structure of this file is described in Section C.1.
- A *type description file* (with extension "*.adt*"), which contains a description of the type of the EVE constant `MAIN` and therefore of the result of the whole program. The structure of this file is described in Section C.1. The type tree has been transformed into this file by converting each node in the tree to a line with a leading number as its identification. The atomic EVE types are represented by single letters:
  - `I` stands for the integer type,
  - `B` stands for the boolean type and
  - `C` stands for the character type.

  Each compound data type is represented by a single letter and a list of numbers that represent the types of the components of this compound type:
  - `L` x stands for a list where the type of the components is represented by the expression in the line x,
  - `T` $x_1 \ldots x_n$ stands for a tuple where the types of the components are represented by the expressions in the lines $x_1, \ldots, x_n$.
  - `U` $x_1 \ldots x_n$ stands for a union where the types of the components are represented by the expressions in the lines $x_1, \ldots, x_n$.

  For example the EVE type `L0` that is defined by the following EVE definitions:

```
T0 == TUPLE OF
       i1 : INT,
       i2 : INT,
       b  : BOOL;

L0 == LIST OF T0;
```

  will be transformed into the following type description file:

```
#0:  1          ;Root of type is in line 1
1:   L  2       ;List. Comps described in line 2
2:   T  3  3  4 ;Tuple. Comps described in 3, 4
3:   I          ;Integer
4:   B          ;Boolean
```

Lexical Structure

Letter	=	`a` | ... | `z` | `A` | ... | `Z` .
Digit	=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` .
Constant	=	*Digit* { *Digit* } .
SignedConst	=	[`+` | `-`] *Constant* .
Ident	=	*Letter* { *Letter* | *Digit* } .
Register	=	*Ident* .
Label	=	*Ident* .
Event	=	*Ident* .
Function	=	*Ident* .
Indir	=	*Label* | *SignedConst* .

Types of Operations

OpReReIn	=	ADD | SUB | MUL | DIV | MOD | AND | OR | EQUAL | LESS | LESSEQ | NEG | NOT | FAILED | COPYI | COPYP | SUBMIT .
OpCoReIn	=	ADDC | SUBC | SUBRC | MULC | DIVC | DIVRC | MODC | MODRC | EQUALC | LESSC | LESSRC | LESSEQC | LESSEQRC | READ .
OpReReCo	=	SEND | WRITEI | WRITEP .
OpCoReCo	=	SENDC | WRITEC .
OpReInIn	=	SWITCH .
OpReIn	=	DELP | CONST | CONTEXT | APPLY | RETURN .
OpInIn	=	SPLIT .
OpCoIn	=	RETURNC | ALLOC .
OpFuIn	=	FASTCALL | TAILCALL .
OpEvIn	=	WAIT .
OpLa	=	JUMP .
Op	=	TERMINATE .

Structure of a Code Line

Line	=	[*Label*] *CommandWithArgs* .
CommandWithArgs	=	*OpReReIn Register Register Indir* | *OpCoReIn Constant Register Indir* | *OpReReCo Register Register Constant* | *OpCoReCo Register Register Constant* | *OpReInIn Register Indir Indir* | *OpReIn Register Indir* | *OpInIn Indir Indir* | *OpCoIn Constant Indir* | *OpFuIn Function Indir* | *OpEvIn Event Indir* | *OpLa Label* | *Op* .

Code

Code = { *Line* } .

Function Code

FctCode	=	`FUNCTION` *Function* `(` *Constant* , *Constant* `)`
		Code
		`END` *Function* .

Program File

PrgFile	=	`PROGRAM` *Ident*
		{ `EXPORT` *Function* { , *Function* } }
		{ *FctCode* }
		`END` *Ident* .

Expression File

ExprFile	=	`EXPRESSION` *Ident*
		{ `IMPORT` *Function* { , *Function* } }
		`BEGIN` `(` *Constant* , *Constant* `)`
		Code
		`END` *Ident* .

Type Description File

Atom	=	`I` | `B` | `C` .
List	=	`L` *Constant* .
Tuple	=	`T` *Constant* { *Constant* } .
Union	=	`U` *Constant* { *Constant* } .
Node	=	*Atom* | *List* | *Tuple* | *Union* .
TypeLine	=	*Constant* : *Node* .
TypeFile	=	`#0: 1` *TypeLine* { *TypeLine* } .

C.2 Semantics

This section gives an informal description of the semantics of the various ADAM instructions. Most of the ADAM assembler instructions are usual two address instructions as the ADAM machine is essentially a controlflow machine. However, there are additional instructions that introduce explicit or implicit parallelism. These instructions are specially emphasized in this description.

The ADAM instructions can be divided into four classes:

1. Arithmetical-Logical Instructions,
2. Control Instructions,
3. Function Call Instructions,
4. Storage Instructions.

As the instructions of different classes have slightly different characteristics, the presentation of the semantics is split according to the above classes. However, to keep a general structure of the description of each instruction for all ADAM instructions the following scheme of an semantic description is used:

- **Instruction:**
- **Description:**
- **Successor Instruction:**

In this section the following typed variables are used:

- c, c' are *Constants*,
- r, r' are *Registers*,
- l, l', l_1, l_2 are *Indirs*,
- ev is an *Event*.

C.2.1 Arithmetical-Logical Instructions

Integer Addition

- **Instruction:** l: ADD r, r', l'
- **Description:** $r \leftarrow r + r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Addition with a Constant

- **Instruction:** l: ADDC c, r, l'
- **Description:** $r \leftarrow r + c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Subtraction

- **Instruction:** l: SUB r, r', l'
- **Description:** $r \leftarrow r - r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Subtraction with a Constant

- **Instruction:** l: SUBC $c,\ r,\ l'$
- **Description:** $r \ \leftarrow \ r - c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Reverse Integer Subtraction with a Constant

- **Instruction:** l: SUBRC $c,\ r,\ l'$
- **Description:** $r \ \leftarrow \ c - r$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Multiplication

- **Instruction:** l: MUL $r,\ r',\ l'$
- **Description:** $r \ \leftarrow \ r * r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Multiplication with a Constant

- **Instruction:** l: MULC $c,\ r,\ l'$
- **Description:** $r \ \leftarrow \ r * c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Division

- **Instruction:** l: DIV $r,\ r',\ l'$
- **Description:** $r \ \leftarrow \ r/r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Division with a Constant

- **Instruction:** l: DIVC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ r/c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Reverse Integer Division with a Constant

- **Instruction:** l: DIVRC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ c/r$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Remainder

- **Instruction:** l: MOD $r,\ r',\ l'$
- **Description:** $r\ \leftarrow\ r\%r'$ (% ... remainder of a division)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Remainder with a Constant

- **Instruction:** l: MODC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ r\%c$ (% ... remainder of a division)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Reverse Integer Remainder with a Constant

- **Instruction:** l: MODRC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ c\%r$ (% ... remainder of a division)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Boolean Conjunction

- **Instruction:** l: AND r, r', l'
- **Description:** $r \leftarrow r \& r'$ (& ... boolean conjunction operation)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Boolean Disjunction

- **Instruction:** l: OR r, r', l'
- **Description:** $r \leftarrow r \mid r'$ (| ... boolean disjunction operation)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Equality

- **Instruction:** l: EQUAL r, r', l'
- **Description:** $r \leftarrow r = r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Equality with a Constant

- **Instruction:** l: EQUALC c, r, l'
- **Description:** $r \leftarrow r = c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Strict Order

- **Instruction:** l: LESS r, r', l'
- **Description:** $r \leftarrow r < r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Strict Order with a Constant

- **Instruction:** l: LESSC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ r < c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Reverse Integer Strict Order with a Constant

- **Instruction:** l: LESSRC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ c < r$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Reflexive Order

- **Instruction:** l: LESSEQ $r,\ r',\ l'$
- **Description:** $r\ \leftarrow\ r <= r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Integer Reflexive Order with a Constant

- **Instruction:** l: LESSEQC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ r <= c$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Reverse Integer Reflexive Order with a Constant

- **Instruction:** l: LESSEQRC $c,\ r,\ l'$
- **Description:** $r\ \leftarrow\ c <= r$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Integer Negation

- **Instruction:** l: NEG r, r', l'
- **Description:** $r \leftarrow -r'$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Boolean Negation

- **Instruction:** l: NOT r, r', l'
- **Description:** $r \leftarrow \neg r'$ ($\neg$ ... boolean negation operation)
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Test for Error Value

- **Instruction:** l: FAILED r, r', l'
- **Description:** $r \leftarrow \begin{cases} \text{TRUE} & \textbf{if} \text{ in } r' \text{ the error flag is set,} \\ \text{FALSE} & \textbf{otherwise} \end{cases}$
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

C.2.2 Control Instructions

Duplicate an Integer Value

- **Instruction:** l: COPYI r, r', l'
- **Description:** $r \leftarrow r'$
  Copy the value of register r' to register r.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Duplicate an Tuple Pointer

- **Instruction:** l: COPYP r, r', l'
- **Description:** $r \leftarrow r'$
  As r' is a tuple pointer in this case its reference count is decreased by this instruction.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Delete a Pointer Value

- **Instruction:** l: DELP r, l'
- **Description:** This instruction deletes the reference r to some tuple. This instruction is needed to handle the garbage collection which is done via reference counting.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Split Control Flow into Parallel Pathes

- **Instruction:** l: SPLIT l_1, l_2
- **Description:** This instruction creates the two parallel threads of control l_1 and l_2 from the incoming thread l. This instruction and the function call are the only possibilities to create parallelism in the ADAM assembler language.
- **Successor Instruction:**
  $\begin{cases} l_1 & \textbf{if } l_1 \in \mathit{Label} \\ l + l_1 & \textbf{if } l_1 \in \mathit{SignedConst} \end{cases}$
  and
  $\begin{cases} l_2 & \textbf{if } l_2 \in \mathit{Label} \\ l + l_2 & \textbf{if } l_2 \in \mathit{SignedConst} \end{cases}$

Switch Control Flow

- **Instruction:** l: SWITCH r, l_1, l_2
- **Description:** This instruction only switches the controlflow to l_1 if r contains the code for TRUE, to l_2 if r contains the code for FALSE and to the next line if r contains an error value. No register values are changed.
- **Successor Instruction:** $\begin{cases} l_1 & \textbf{if } r = \mathrm{TRUE} \\ l_2 & \textbf{if } r = \mathrm{FALSE} \\ l + 1 & \textbf{if } r = \mathrm{ERROR} \end{cases}$

Route Control Flow

- **Instruction:** l: JUMP l'
- **Description:** Perform an explicit jump to the label l'. Note: For backward jumps only this instruction can be used (i.e. no indirect backward jumps are possible).
- **Successor Instruction:** l'

Wait for a Synchronization Event

- **Instruction:** l: WAIT ev, l'
- **Description:** This instruction is used to synchronize parallel threads of control. The event ev can be viewed as a toggle that indicates if a WAIT on this event has already been performed. If yes the toggle is reset and the computation continues at line l', thereby joining two threads into one. If no the toggle is set and the controlflow is suspended until another thread reaches a WAIT instruction on this event.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Generate a Long Constant

- **Instruction:** l: CONST r, l'
- **Description:** This instruction takes the constant that stands in the next line and puts it into the register r.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

C.2.3 Function Call Instructions

Request Standard Function Activation

- **Instruction:** l: FASTCALL f, l'
- **Description:** This instruction initiates the computation of the function f. First an activation record will be allocated and a pointer to it will arrive at the line $l+1$. The result of the computation of the function will arrive at the line $l+2$.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Request Tail-Recursive Function Activation

- **Instruction:** l: TAILCALL f, l'
- **Description:** This instruction initiates a tail recursive computation of the function f. The own context of the current function has to be sent to the function activation of f such that the result can be directly sent to the caller of the current function. This is done with the CONTEXT instruction. A pointer to the activation record will appear at the line $l+1$.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Submit the Current Result Address to Callee

- **Instruction:** l: CONTEXT $r,\ l'$
- **Description:** This instruction submits the result address, which is stored inside the current function activation, to the newly created activation to which r points. Together with TAILCALL this is used to shortcut the returning process for tail recursive functions.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Request activation of a non-fixed function

- **Instruction:** l: APPLY $r,\ l'$
- **Description:** This instruction takes a function address in r and initiates the computation of the function. First an activation record will be allocated and a pointer to it is returned in the line $l+1$. Hence, APPLY does essentially the same as FASTCALL, but it also operates on higher order functions.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Send Argument to a Function Activation

- **Instruction:** l: SEND $r,\ r',\ i$
- **Description:** Send the register r as the i-th argument to the function activation to which r' points. This r' must be the result of a FASTCALL or of an APPLY instruction.
- **Successor Instruction:** $l + 1$

Send Constant Argument to a Function Activation

- **Instruction:** l: SENDC $c,\ r',\ i$
- **Description:** Send the constant c as the i-th argument to the function activation to which r' points. This r' must be the result of a FASTCALL or of an APPLY instruction.
- **Successor Instruction:** $l + 1$

Request Submission of Curried Function Arguments

- **Instruction:** l: SUBMIT $r,\ r',\ l'$
- **Description:** This instruction delivers the curried arguments that are stored in the curried expression to which r points, to the function activation to which r' points.

- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Send Result Value back to Caller

- **Instruction:** l: RETURN r, l'
- **Description:** This instruction returns the value in the register r as the result of the evaluation of the current function.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Send Constant as Result Value back to Caller

- **Instruction:** l: RETURNC c, l'
- **Description:** This instruction returns the constant c as the result of the evaluation of the current function.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Terminate Actual Activation

- **Instruction:** l: TERMINATE
- **Description:** This instruction terminates the activation of the current function and frees the memory that has been allocated by it.
- **Successor Instruction:** None

C.2.4 Storage Instructions

Allocate Tuple

- **Instruction:** l: ALLOC c, l'
- **Description:** This instruction allocates a tuple of the size c. A pointer to this tuple will arrive at the line $l + 1$. This instruction performs a side-effect.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l + l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Read Contents of a Tuple Field

- **Instruction:** l: READ $c,\ r,\ l'$
- **Description:** This instruction reads the value of the c-th component of the tuple to which r points (counting starts from 0). The value of this component will arrive at the line $l+1$. This instruction performs a side-effect.
- **Successor Instruction:** $\begin{cases} l' & \textbf{if } l' \in \mathit{Label} \\ l+l' & \textbf{if } l' \in \mathit{SignedConst} \end{cases}$

Write Integer into Tuple Field

- **Instruction:** l: WRITEI $r,\ r',\ c$
- **Description:** This instruction writes the integer value of register r into the c-th field of the tuple to which r' points. A signal output of this instruction will arrive at the line $l+1$. This instruction performs a side-effect.
- **Successor Instruction:** $l+1$

Write Pointer into Tuple Field

- **Instruction:** l: WRITEP $r,\ r',\ c$
- **Description:** This instruction writes the pointer value of register r into the c-th field of the tuple to which r' points. The difference to the WRITEI instruction is that here the reference counter of the tuple to which r points is decreased. This only affects the garbage collection. A signal output of this instruction will arrive at the line $l+1$. This instruction performs a side-effect.
- **Successor Instruction:** $l+1$

Write Integer Constant into Tuple Field

- **Instruction:** l: WRITEC $c',\ r',\ c$
- **Description:** This instruction writes the integer constant value c' into the c-th field of the tuple to which r' points. A signal output of this instruction will arrive at the line $l+1$. This instruction performs a side-effect.
- **Successor Instruction:** $l+1$

Appendix D

EVE Test Programs

This appendix contains some EVE programs that have been used to make benchmarks. However, the circular prime number generators that are presented in `CircPrime.eve` are not used for benchmarking but for explaining in more detail the differences of the parallel behaviour of two similar programs that only differ in the data dependencies inside the programs. The algorithms themselves and their runtime behaviour as well as the behaviour of the two circular prime number generators are discussed in Chapter 5. Table D.1 shows a list of those EVE program files that are included in this appendix.

Program Name	*Description of the Program*	*Page*
`Fib.eve`	Fibonacci Number Program	211
`QuickSort.eve`	Quick Sort Program	212
`Queens.eve`	Program for Solving N-Queens Problem	214
`Prime.eve`	Straightforward Prime Number Generator	217
`CircPrime.eve`	Two Circular Prime Number Generators	219

Table D.1: EVE programs

Fib.eve Page: 1

```
-- Fib.eve;
--
-- -------------------------------------------------------------------------
--
-- Simple Fibonacci Function.
--
-- Problem Specification:
--  Given: n, n ... integer.
--  Find:  fib(n) s.t.
--               fib(0) = 1,
--               fib(1) = 1,
--               fib(n) = fib(n-2) + fib(n-1).
--
-- -------------------------------------------------------------------------

Fib[n] : INT[INT] ==
  IF n<=1
    THEN 1
    ELSE Fib[n-1]+Fib[n-2] ;

-- -------------------------------------------------------------------------
-- Result expression
-- -------------------------------------------------------------------------

MAIN : INT == Fib[18];
```

QuickSort.eve Page: 1

```
-- QuickSort.eve;
--
-- -----------------------------------------------------------------------
--
-- Eve Program for performing the Quicksort algorithm.
--
-- Problem Specification:
--  Given: l, l ... list of integers of the form [l_1, ..., l_n].
--  Find:  l', l' ... list of integers of the form [l_1, ..., l_n] s.t.
--              for all 1<=i<n: l'_i <= l'_{i+1},
--              exists f : [1..n] -> [1..n] : l_i = l'_{f(i)},
--                                            f is bijective.
--
-- -----------------------------------------------------------------------

-- ------------------------------------------------------------------------
-- Type Definitions
-- ------------------------------------------------------------------------

IntList == LIST OF INT ;

-- ------------------------------------------------------------------------
-- Basic list operations
-- ------------------------------------------------------------------------

Append[l1, l2] : IntList[IntList, IntList] ==
 IF NULL[l1]
   THEN l2
   ELSE { hd : INT     == l1.HEAD ;
          tl : IntList == l1.TAIL }
        hd :: Append[tl,l2] ;

-- ------------------------------------------------------------------------
-- FromToStep[a,b,limit] creates the same list as the list comprehension
-- [a,b..limit], namely  [a,b,b+(b-a),...,limit].
-- ------------------------------------------------------------------------

FromToStep[a,b,limit] : IntList[INT,INT,INT] ==
 { inc  : INT == b-a ;
   test[x,y] : BOOL[INT,INT] == IF a<b
                                  THEN IGT[x,y]
                                  ELSE ILT[x,y] ;
   fs[a] : IntList[INT] == IF test[a,limit]
                             THEN IntList[]
                             ELSE a :: fs[a+inc] }
 fs[a] ;

-- ------------------------------------------------------------------------
-- Functions
-- ------------------------------------------------------------------------
```

```

QSort[l] : IntList[IntList] ==
  IF NULL[l]
    THEN IntList[]
    ELSE {
          -- --------------------------------------------------------------
          -- Result of splitting a list with the ‘Part’ function.
          -- --------------------------------------------------------------

          PTuple == TUPLE OF
                      less            : IntList,
                      greater         : IntList,
                      pivot           : INT;

          -- --------------------------------------------------------------
          -- Splitting a list ‘l’ in a list with all elements < ‘x’
          -- and a list with all elements > ‘x’.
          -- --------------------------------------------------------------

          Part [l,x] : PTuple[IntList, INT] ==
            IF NULL[l]
              THEN PTuple[IntList[], IntList[], x]
              ELSE { hd : INT       == l.HEAD ;
                     tl : IntList   == l.TAIL ;
                     p  : PTuple    == Part[tl,x];
                     pl : IntList   == p.less ;
                     pg : IntList   == p.greater }
                   IF hd <= x
                     THEN PTuple[hd :: pl, pg, x]
                     ELSE PTuple[pl, hd :: pg, x] ;

          -- --------------------------------------------------------------
          -- Local variables
          -- --------------------------------------------------------------

          HD : INT       == l.HEAD ;
          TL : IntList   == l.TAIL ;
          p  : PTuple    == Part[TL,HD] ;
          l1 : IntList   == QSort[p.less] ;
          l2 : IntList   == QSort[p.greater]                             }
         Append[l1,Append[IntList[HD],l2]];

-- ---------------------------------------------------------------------------
-- Result expression
-- ---------------------------------------------------------------------------

MAIN : IntList ==
 { l  : IntList == FromToStep[50,49,1]  }
 QSort[l];
```

```
Queens.eve                                                          Page:   1
-- Queens.eve;
--
-- -------------------------------------------------------------------------
--
-- Program for solving the 8-Queens problem.
-- First order version (i.e. this program doesn't use higher order fcts)
--
-- Problem Specification:
--  Given: n, n ... integer
--  Find:  b, b ... list of boards of the form [b_1,...,b_m]  s.t.
--              for all 1<=i<=m : the board b_i contains n queens,
--                                no queen checks another queen on board b_i
--
-- -------------------------------------------------------------------------

-- -------------------------------------------------------------------------
-- Type Definitions
-- -------------------------------------------------------------------------

Board       == LIST OF INT;
BoardList   == LIST OF Board;

IntList     == LIST OF INT;

-- -------------------------------------------------------------------------
-- Basic list operations
-- -------------------------------------------------------------------------

Append[l1, l2] : IntList[IntList, IntList] ==
 IF NULL[l1]
   THEN l2
   ELSE { hd : INT    == l1.HEAD ;
          tl : IntList  == l1.TAIL }
        hd :: Append[tl,l2] ;

Append0[l1, l2] : BoardList[BoardList, BoardList] ==
 IF NULL[l1]
   THEN l2
   ELSE { hd : Board      == l1.HEAD ;
          tl : BoardList  == l1.TAIL }
        hd :: Append0[tl,l2] ;

Length[l] : INT[IntList] ==
 IF NULL[l]
   THEN 0
   ELSE 1+Length[TAIL[l]] ;

-- -------------------------------------------------------------------------
-- Functions
-- -------------------------------------------------------------------------
```

```

Queens[size] : BoardList[INT] ==
 {
  -- -----------------------------------------------------------------------
  -- Test whether it is safe to put a queen on the position ‘row’/‘col’,
  -- where ‘bs’ is a board on which queens have been placed only on the
  -- columns 1,..,‘col’-1.
  -- -----------------------------------------------------------------------

  Allsafe[col,row,bs] : BOOL[INT,INT,Board] ==
   IF NULL[bs]
     THEN TRUE
     ELSE {
           hd : INT   == HEAD[bs];
           tl : Board == TAIL[bs];

           dx : INT  == Length[bs];
           dy : INT  == row-hd;
          }
          (hd#row) & (ABS[dx]#ABS[dy]) & Allsafe[col,row,tl] ;

  -- -----------------------------------------------------------------------
  -- This function returns a list of that valid boards with queens on the
  -- columns 1,...,‘col’ that can be derived from the board ‘b’ (with
  -- queens on the columns 1,...,‘col’-1).
  -- -----------------------------------------------------------------------

  Add1[col,row,b] : BoardList[INT,INT,Board] ==
   IF row>=size+1
     THEN BoardList[]
     ELSE { rest : BoardList  == Add1[col,row+1,b] }
          IF Allsafe[col,row,b]
            THEN Append[b,Board[row]] :: rest
            ELSE rest ;

  -- -----------------------------------------------------------------------
  -- This function takes a list of all valid boards with queens on the
  -- columns 1,...,‘col’ where ‘bss’ is a list of all valid boards with
  -- queens on the columns 1,...,‘col’-1.
  -- -----------------------------------------------------------------------

  Add[col,bss] : BoardList[INT,BoardList] ==
   IF NULL[bss]
     THEN BoardList[]
     ELSE {
           b    : Board      == HEAD[bss];
           bs   : BoardList  == TAIL[bss];
          }
          Append0[ Add1[col,1,b] , Add[col,bs] ];

```

```
Queens.eve                                                      Page:   3
  -- ---------------------------------------------------------------------
  -- This function computes a list of all valid boards with queens on the
  -- columns 1,...,'col'.
  -- ---------------------------------------------------------------------

  QueensCol[col] : BoardList[INT] ==
   IF col=0
     THEN BoardList[Board[]]
     ELSE { boards : BoardList == QueensCol[col-1] }
          Add[col,boards];
 }
 QueensCol[size];

-- -----------------------------------------------------------------------
-- Result expression
-- -----------------------------------------------------------------------

MAIN : BoardList ==  Queens[6];
```

```
-- Prime.eve;
--
-- ------------------------------------------------------------------------
--
-- Program for generating a list of prime numbers up to a given number.
--
-- Problem Specification:
--  Given: n, n ... integer > 0
--  Find:  l, l ... list of the form [l_1, ..., l_m]   s.t.
--             for all 1<=i<=m:  l_i is prime, l_i < l_{i+1},
--             for all p: p is prime, p < n ==>
--                        exists j, 1<=j<=m: l_j = p.

-- ------------------------------------------------------------------------
-- Type Definitions
-- ------------------------------------------------------------------------

IntList   == LIST OF INT;
PredOfInt == BOOL[INT];

-- ------------------------------------------------------------------------
-- Functions
-- ------------------------------------------------------------------------

-- ------------------------------------------------------------------------
-- IsPrime[x]:
--  Test whether the integer `x' is prime by dividing by 2 and by all odd
--  numbers up to the square root of `x'.
-- ------------------------------------------------------------------------

IsPrime[x] : PredOfInt ==
 IF (x=2) | (x=3)
   THEN TRUE
   ELSE { test[x,n] : BOOL[INT,INT] ==
            IF n*n > x
              THEN TRUE
              ELSE IF x % n = 0
                     THEN FALSE
                     ELSE test[x,n+2]
          }
          (x%2#0) & test[x,3];

-- ------------------------------------------------------------------------
-- PrimesTo[n]:
--  Generate a list of all prime numbers up to the number `n'.
--  First order version (no higher order functions used).
-- ------------------------------------------------------------------------

PrimesTo[n] : IntList[INT] ==
 { Aux[m] : IntList[INT] ==
```

```
Prime.eve                                                            Page:   2
     IF m>n
       THEN IntList[]
       ELSE IF IsPrime[m]
              THEN m :: Aux[m+2]
              ELSE      Aux[m+2] ;
 }
 IF n<2
   THEN IntList[]
   ELSE 2 :: Aux[3] ;

-- ----------------------------------------------------------------------------
-- Result expression
-- ----------------------------------------------------------------------------

MAIN : IntList == PrimesTo[170];
```

```
CircPrime.eve                                                    Page:   1
-- CircPrime.eve;
--
-- ---------------------------------------------------------------------------
--
-- Circular Algorithms for computing the list of all prime numbers.
-- This file contains implementations of 2 different, circular prime
-- number algorithms:
--
--   1)     "Introduction to Functional Programming using RUFL",
--                         E. P. Wentworth, p.87
--
--   2)   "Adam & Eve - An Abstract Dataflow Machine and its Programming
--                                Language",
--                             Master’s Thesis,
--                               W. Schreiner
--
-- ---------------------------------------------------------------------------
--  Data Structures
-- ---------------------------------------------------------------------------

IntList   == LIST OF INT;
PredOfInt == BOOL[INT];

-- ---------------------------------------------------------------------------
-- Algorithm of Peter Wentworth
-- ---------------------------------------------------------------------------

-- ---------------------------------------------------------------------------
-- Test whether a number ‘n’ is prime by dividing it by all prime numbers
-- less than the sqare root of ‘n’. These primes are taken from the
-- list of all prime numbers in ‘primes’.
-- ---------------------------------------------------------------------------

isPrime[n,primes] : BOOL[INT,IntList] ==
 { p  : INT     == HEAD[primes];
   ps : IntList == TAIL[primes] }
 IF n<(p*p)
   THEN TRUE
   ELSE IF (n % p) = 0
          THEN FALSE
          ELSE isPrime[n,ps]  ;

-- ---------------------------------------------------------------------------
-- Compute a list of all odd prime numbers from ‘n’ to ‘limit’.
-- ‘primes’ contains a list of all prime numbers.
-- ---------------------------------------------------------------------------

oddPrimesFromTo[n,limit,primes] : IntList[INT,INT,IntList] ==
 IF n>limit
   THEN IntList[]
```

```
CircPrime.eve                                                       Page:   2
   ELSE IF isPrime[n,primes]
          THEN n :: oddPrimesFromTo[n+2,limit,primes]
          ELSE      oddPrimesFromTo[n+2,limit,primes] ;


-- --------------------------------------------------------------------------
-- Algorithm of Wolfgang Schreiner
-- --------------------------------------------------------------------------

-- --------------------------------------------------------------------------
-- Test whether 'n' is prime and add it to 'r' iff it is prime.
-- --------------------------------------------------------------------------

cons[n,r,primes] : IntList[INT,IntList,IntList] ==
 { p  : INT     == HEAD[primes];
   ps : IntList == TAIL[primes]
 }
 IF n < p*p
   THEN n :: r   ELSE
 IF n % p = 0
   THEN r
 ELSE cons[n,r,ps];

-- --------------------------------------------------------------------------
-- Compute a list of all odd prime numbers from 'n' to 'limit'.
-- 'primes' contains a list of all prime numbers.
-- --------------------------------------------------------------------------

sieve[n,limit,primes] : IntList[INT,INT,IntList] ==
  IF n>limit
    THEN IntList[]
    ELSE { r : IntList == sieve[n+2,limit,primes] }
         cons[n,r,primes] ;

-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-- --------------------------------------------------------------------------
-- MAIN part of Alg. of P. Wentworth
-- --------------------------------------------------------------------------

-- MAIN : IntList == 2 :: oddPrimesFromTo[3,70,MAIN] ;

-- --------------------------------------------------------------------------
-- MAIN part of Alg. of W. Schreiner
-- --------------------------------------------------------------------------

MAIN : IntList == 2 :: sieve[3,70,MAIN];
```

Appendix E

Definition Modules of the EVE Compiler

This appendix contains the documented definition modules of all parts of the prototype EVE compiler. These definition modules also form the documentation of the data types and procedures that are exported by the data encapsulations for the abstract data types that are needed by the compiler. Some of these procedures are used in Appendix A, describing the syntax analysis and in Appendix B, describing the code generation.

As it has already been described in Chapter 4 all the modules of the EVE compiler are separated into three classes:

1. The first class describes the top level module of the whole compiler and the **top level modules** of the several tasks of the compiler. Table E.1 shows the modules contained in this class.

2. The second class describes the **data encapsulations** that are needed by the compiler. Table E.2 shows the modules contained in this class.

3. The third class describes some **special modules** especially for I/O. Table E.3 shows the modules contained in this class.

The above mentioned figures also show the modules in the same order as they appear in the following sections.

Module Name	*Description of the Module*	*Page*
`EveCompile`	Top level module of the whole compiler	223
`EveLexAn`	Module for lexical analysis	224
`EveSyntAn`	Module for syntax analysis	230
`EveLambdaLift`	Module for λ-lifting	231
`EveCodeGen`	Module for code generation	232

Table E.1: Top Level Modules

Module Name	*Abstract Data Type*	*Page*
`EveAST`	Abstract syntax tree	234
`EveCode`	Code area	247
`EveEnvironment`	Environment for code generation	254
`EveEvent`	Events for code generation	257
`EveLabel`	Labels for code generation	260
`EveList`	Lists of numbers	263
`EveListList`	Lists of lists of numbers	268
`EveRegister`	Registers for code generation	272
`EveStringLib`	Strings	275
`EveSym`	Symbol table and symbol list	277
`EveType`	EVE data types	287

Table E.2: Data Encapsulations

Module Name	*Description*	*Page*
`EveCoIO`	Handling of result files	295
`EveError`	Error handling	298
`EveGetChar`	Elementary input	299
`EveWrite`	Output of enumerations and sets	300

Table E.3: Special Modules

E.1 Top Level Modules

```
EveCompile                                                      Page:   1

DEFINITION MODULE EveCompile;     (* Top Level module of the Eve Compiler
=========================================================================
 This module contains as its only procedure the top level procedure for
 the whole compilation process. A call to this procedure is the only
 thing that is done in the module 'Eve.MOD'.
 ------------------------------------------------------------------------  *)


TYPE
    errorType = (none, CommandLineError, LexicalError, SyntaxError,
                 LambdaLiftError, SemanticError);

VAR
    success : BOOLEAN;   (* states if the parsing was successful *)
    error   : errorType; (* states which error has occured *)

PROCEDURE Compile;       (* Top Level procedure; called in 'Eve.MOD' *)

END EveCompile.
```

```
EveLexAn                                                        Page:   1

DEFINITION MODULE EveLexAn;                          (* Lexical Analyzer
========================================================================
 Top Level module of the lexical analysis (part of 1st pass).
 All objects from the lexical analyzer that are used by the syntax
 analyzer are described here. The procedures that are exported by
 the lexical analyzer can be split into several categories:

   - One routine gets the next symbol from the program.

   - One class of procedures performs the task of renaming variables.

   - Direct access routines fetch the real name of a variable that is
     represented by a CARDINAL number.

   - A routine for checking wether an object is predefined or not is
     also needed.

   - Several routines for explicit memory deallocation allow it to
     free some memory after the first pass of the compiler.

   - Several routines for printing information about a variable,
     string or such things are mainly used for testing.

 ---------------------------------------------------------------------- *)

TYPE Symbol = (
 (* special *)  noSy, eofSy,
 (* keywords *) caseSy, elseSy,
                ifSy, listSy, ofSy,
                thenSy, tupleSy, unionSy,
 (* classes *)  char, ident, number, string,
 (* arithm. *)  divSy, minusSy, modSy, plusSy, timesSy,
 (* boolean *)  andSy, notSy, orSy,
 (* rel.ops *)  eqSy, geSy, gtSy, leSy, ltSy, neSy,
 (* brackets*)  lbraceSy, lbracketSy, lparSy,
                rbraceSy, rbracketSy, rparSy,
 (* others  *)  colonSy, commaSy, conversionSy, defineSy, doublecolonSy,
                selectionSy, semicolonSy);

CONST SymbolX =                         (* Needed for printing a symbol *)
  "noSy,eofSy,caseSy,elseSy,"+
  "ifSy,listSy,ofSy,"+
  "thenSy,tupleSy,unionSy,"+
  "char,ident,number,string,divSy,minusSy,modSy,plusSy,timesSy,"+
  "andSy,notSy,orSy,eqSy,geSy,gtSy,leSy,ltSy,neSy,"+
  "lbraceSy,lbracketSy,lparSy,rbraceSy,rbracketSy,rparSy,"+
  "colonSy,commaSy,conversionSy,defineSy,doublecolonSy,"+
  "selectionSy,semicolonSy ";

(* ---------------------------------------------------------------------*)
```

EveLexAn Page: 2

```

CONST
      maxNrTransFcts = 29;         (* For translation of fctcalls to      *)
                                   (* explicit nodes in the AST (2nd pass).*)
      MaxInt = MAX(INTEGER) DIV 2; (* Max value of a constant of          *)
                                   (* type INT.                           *)

TYPE
     TransFctNames = ARRAY [1..maxNrTransFcts] OF CARDINAL;
                      (* Names of fcts that have to be translated to       *)
                      (* explicit nodes in the AST (2nd pass).             *)

                              (* The following variables are READ-ONLY !! *)

VAR sy     : Symbol;     (* the kind of the symbol detected by the last     *)
                         (* recent call of NewSym                           *)
    col    : CARDINAL;   (* the column number, where the symbol 'sy' starts *)
                         (* in the printed output                           *)
    spix   : CARDINAL;   (* 'spelling index' of the last recently detected  *)
                         (* identifier                                      *)
    value  : CARDINAL;   (* the value of the last recently detected number  *)
    saddr  : CARDINAL;   (* the start address of the last recently detected *)
                         (* string in the string list                       *)
    slen   : CARDINAL;   (* the length of the last recently detected string *)
    charVal: CHAR;       (* the character value of last recently detected   *)
                         (* character; after having detected a char. also   *)
                         (* saddr points to the place where charVal has been*)
                         (* entered to the string storage; slen=1           *)

      (* The following names have to be known outside the lexical analyzer *)
      (* too. Hence, they are explicitly exported.                         *)

    nameOfHEAD, nameOfTAIL,
    nameOfABS, nameOfODD, nameOfCAP,
    nameOfNULL, nameOfFAILED, nameOfORD, nameOfMAIN,
    nameOfTRUE, nameOfFALSE,
    nameOfMAXINT, nameOfMININT,
    nameOfERROR
                      : CARDINAL;    (* spelling indices of the predefined *)
                                     (* fcts that stay present in the AST. *)
                                     (* READ ONLY!                         *)

    noName            : CARDINAL;    (* READ ONLY! The value               *)
                                     (* is set in a local module *)

    transFctNames     : TransFctNames;
                           (* A list of fct-names that must be translated  *)
                           (* into nodes of the AST by the procedure:      *)
                           (* EveAST.Translate                             *)

```

```
(* ---------------------------------------------------------------------- *)

PROCEDURE NewSym;  (* Central procedure in the lexical analysis.
                      It fetches the next symbol from the EVE source
                      program and sets the following global variables
                      in this module:
                       sy, col, spix, value, saddr, slen, charVal.         *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*    Procedures for handling the renaming of variables in the program.   *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE Rename (VAR name : CARDINAL; level : CARDINAL);
 (*
   Input:  name  ... The name of a variable in the EVE program.
           level ... The level where this variable was defined.
   Output: name ... A new, unique name that represents the same object
                    from now on.

   By this procedure, a variable is renamed if there already exists a
   different object of the same name in the current scope. The renaming
   ranges over the whole 'level' and has to be explicitly undone with a
   call to 'UnRename'.                                                     *)

PROCEDURE UnRename (name : CARDINAL);
 (*
   Input: name ... The name of a variable whose renaming has to be undone.

   This procedure deletes the last renaming if it affected 'name'. If the
   last renaming was not applied to 'name' an error occurs.                *)

PROCEDURE UnRenameLevel (level : CARDINAL);
 (*
   Input: level ... The static nesting level for which all renamings have to
                    be undone.

   This procedure deletes all renamings for objects in the nesting
   evel 'level'. This is usually needed at the end of a function or
   of a block.                                                             *)

PROCEDURE DisableRenaming;
 (*
   Turns the renaming mechanism off. This is needed for the correct
   handling of selections from tuples and conversions of tuples as
   in the following case obviously the field x and not the variable x is
   meant in the expression of the local definition:

              { Alpha == TUPLE OF
```

```
EveLexAn                                                            Page:   4

                           x : INT,
                           y : BOOL;
                {
                  x : BOOL  == TRUE ;
                  a : Alpha == Alpha[1,TRUE]
                }
                a.x
              }                                                            *)

PROCEDURE EnableRenaming;
 (*
   Turns the renaming mechanism on. This is the default status of the
   renaming. Typically, the renaming is just turned off for one or two
   statements, then it is turned on again.                                 *)

PROCEDURE GetVirtualName (name : CARDINAL) : CARDINAL;
 (*
   Input:  name ... The name of a variable.
   Output: name ... The ambigous name that was renamed and yielded the
                    value that was the input to this procedure.

   This procedure inverses the renaming process for 'name' without
   undoing the renaming.                                                   *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*  Procedures for accessing the local namelist from outside               *)
(*  the lexical analyzer.                                                  *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE FetchNameStr (spix : CARDINAL; VAR name : ARRAY OF CHAR);
 (*
   Input:  spix ... An Eve object, represented by a spelling index of
                    the local namelist.
   Output: name ... The name of the object as a string like it appears
                    in the Eve code.

   This routine is mainly needed for testing.                              *)

PROCEDURE FetchName(VAR spix : CARDINAL; name : ARRAY OF CHAR);
 (*
   Input:  name ... The name of the object as a string like it appears
                    in the Eve code.
   Output: spix ... An Eve object, represented by a spelling index of
                    the local namelist.

   The other direction compared to 'FetchNameStr'.                         *)

(* ----------------------------------------------------------------------- *)
```

```
EveLexAn                                                         Page:   5

(*                                                                      *)
(*          Some operations on the names in an Eve program              *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE isPredefined (spix : CARDINAL) : BOOLEAN;
 (*
   Input:  spix  ... The name of an Eve object.
   Output: TRUE  ... if spix is a predefined object (like CONS, CAP, ... )
           FALSE ... if spix is not a predefined object (usually).        *)

PROCEDURE GetNewIdent (VAR name : CARDINAL);
 (*
   Output: name ... A new, unique name for an Eve object.

   This procedure is only needed in 'EveSym.NewIdent' where a new, anonymous
   identifier has to be added to the symbol table.                      *)

(* -------------------------------------------------------------------- *)
(*                                                                      *)
(*        Procedures for explicitly discarding local objects.           *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE DisposeNames;
 (*
   This procedure explicitly frees the local namelist that is not used after
   the 1st pass (lexical & syntax analysis) of the compiler any more.     *)

PROCEDURE DisposeStringStorage;
 (*
   This procedure explicitly frees the local stringstorage that is not used
   after the 1st pass (lexical & syntax analysis) of the compiler any more *)

(* -------------------------------------------------------------------- *)
(*                                                                      *)
(*                Procedures for testing the compiler                   *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE PrintName(spix : CARDINAL);
 (*
   Print the real name (string) of the identifier that is denoted
   by 'spix'.                                                           *)

PROCEDURE PrintNameList;
 (*
   Print the real names (strings) of all identifiers that have occured
   in the EVE program so far.                                           *)
```

```
EveLexAn                                                        Page:   6

PROCEDURE PrintSymbol;
 (*
   Print the current contents of the global variables:
     sy, col, spix, value, saddr, slen.
   If the symbol represents an identifier its real name is printed.
   If the symbol represents a string its contents is printed.           *)

PROCEDURE FetchString(saddr, slen : CARDINAL; VAR str : ARRAY OF CHAR);
 (*
   Copy the EVE string that is denoted by the values of ‘saddr’, ‘slen’
   into ‘str’.                                                          *)

PROCEDURE PrintStringList;
 (*
   Print a list of all strings that have occured in the EVE program so far.*)

END EveLexAn.
```

EveSyntAn Page: 1

```
DEFINITION MODULE EveSyntAn;                (*   Pass 1: Syntax Analyzer
==========================================================================
 This module contains the syntax analyzer of the Eve compiler with the
 top level procedure. The modules that are needed by the syntax analyzer are:

   EveAST, EveSym, EveType, EveLexAn,
   EveIO, EveCompile, EveWrite, EveStringLib, EveError.
 ------------------------------------------------------------------------ *)

FROM EveAST IMPORT ASType;

PROCEDURE SyntAn (VAR x : ASType);
 (*
   Top level procedure for the syntax analysis.
   When this procedure is called the given program will be checked for
   lexical (EveLexAn) and syntactical (EveSyntAn) correctness.
   The main result of this step is 'x', and abstract syntax tree that
   represents the given program. Furthermore, some encapsulated information
   will be produced as output of this step:

    - In a symbol table ('EveSym') all objects in the source program with
      their attributes are stored.

    - In the data encapsulation 'EveTypes' all user defined types from the
      source program are stored. These types are only referred to by
      an attribute in the element of the symbol table.                  *)

END EveSyntAn.
```

```
DEFINITION MODULE EveLambdaLift;       (*  Lambda Lifting (part of Pass 2)
==========================================================================
 Top level module for the lambda lifting transformation on the AST
 that is the output of the syntax analysis. The modules that are
 needed in this module are:

   EveAST, EveList, EveListList, EveSym, EveIO.

 Furthermore, this module contains a local module for performing the
 computation of a transitive closure.
 ------------------------------------------------------------------------- *)

FROM EveAST      IMPORT ASType;
FROM EveList     IMPORT List;
FROM EveListList IMPORT ListList;

PROCEDURE LambdaLift (ast : ASType);
 (*
   Procedure performing a lambda lifting on the abstract syntax tree `ast'.
   In this transformation all local function definitions are brought
   on the top level. Global variables in these function definitions
   are turned into additional parameters of the function. The result
   is a set of function definitions on top level that contain no
   global variables and have no local function definitions
   (they are usually called ``super combinators'').
   The liftes functions will be afterwards at the beginning of the
   definition sequence that represents the program (in the order of their
   apearance in the given program).                                       *)

END EveLambdaLift.
```

EveCodeGen Page: 1

```
DEFINITION MODULE EveCodeGen;                  (*  Pass 3: Code Generation
==========================================================================
 Top level module for the code generation (3rd pass) in the Eve compiler.
 The actual code generation is performed by a call to the procedure
 'CodeGen'. The other procedure is used for collecting statistical
 information about the generated code. Both procedures are called
 from 'EveCompile'. The modules that are needed in this module are:

   EveAST, EveCode, EveEnvironment, EveEvent, EveLabel, EveRegister,
   EveList, EveSym, EveLexAn, EveError, EveIO.
 ------------------------------------------------------------------------- *)

FROM EveAST  IMPORT ASType;
FROM EveList IMPORT List;

PROCEDURE CodeGen (ast : ASType; boundVars : List);
 (*
   This procedure takes an abstract syntax tree 'ast' and a list 'boundVars'
   of all top level functions in this tree. It produces Adam assembler
   code. This code is encapsulated by the data encapsulation 'EveCode'.
   This procedure actually produces code for a program file ('.asm'
   extension) and for an expression file ('.ase' extension).
   The former contains the code for all functions in the given program.
   The names of all these functions are exported from the program file.
   The latter contains just the code for the expression that is bound
   to 'MAIN' in the given program. The value of this expression is then
   the result of the whole program.
   See also the comments in 'EveCoIO on the produced files.              *)


PROCEDURE StatisticsOfCodeGen (VAR heapBeginning, heapDispose, heapBefCodeGen,
                               heapAftCodeGen, usedHeap : LONGCARD;
                               VAR usedHeapInPercent : CARDINAL);
 (*
   This procedure collects some statistical information about the
   generated Adam assembler code. The meaning of the parameters, which
   are all output parameters, is:

    heapBeginning ... Available heap at the beginning of the procedure
                      'CodeGen'.
    heapDispose ... Available heap after having discarded data structures
                    of previous passes that are not needed any more.
    heapBefCodeGen ...Available heap immediately before calling the
                      procedure that traverses the AST and created code.
                      The difference to 'heapDispose' is that in the
                      meantime the code area has been initialized the
                      head and the export list have been entered into the
                      code area and two lists that contained the
                      export list and their arities have been discarded.
    heapAftCodeGen ...Available heap after calling this
```

```
EveCodeGen                                                        Page:   2

                      AST traversing procedure and after splitting the
                      code into program and an expression part.
    usedHeap          ... heapCodeGen-heapAftCodeGen.
    usedHeapInPercent ... 100*usedHeap / heapBefCodeGen.                   *)

END EveCodeGen.
```

E.2 Data Encapsulations

```
EveAST                                                           Page:   1

DEFINITION MODULE EveAST;   (*  Data Encapsulation of Abstract Syntax Tree
==========================================================================
 This module contains the opaque data type of an abstract syntax tree
 and all available procedures that work on this data type.
 ------------------------------------------------------------------------ *)

FROM EveList     IMPORT List;
FROM EveListList IMPORT ListList;
FROM EveType     IMPORT EType;


TYPE
      ASType;         (* Encapsulated data type of an abstract syntax tree *)

CONST
      NodeKindX = "programN,defseqN,defN,applyN,callN,curryN,lambdaN,"+
                  "letN,ifN,caseN,composeN,selectN,"+
                  "eqN,ltN,leN,gtN,geN,neN,"+
                  "minusN,plusN,orN,timesN,divN,modN,andN,notN,"+
                  "minusSignN,nilN,bodyN,elseN,lheadN,lbodyN,"+
                  "charN,idN,numberN ";

TYPE
      NodeKind = (programN,defseqN, defN,
                  applyN, callN, curryN, lambdaN,
                  letN, ifN, caseN, composeN, selectN,
                (* Relations *)
                  eqN, ltN, leN, gtN, geN, neN,
                (* Basic Operations *)
                  minusN, plusN, orN, timesN, divN, modN, andN, notN,
                (* Special *)
                   minusSignN, nilN,
                   bodyN, elseN, lheadN, lbodyN,
                (* Atomar *)
                  charN, idN, numberN);

      Flags = (pointer, int, none);    (* Information if value of subtree  *)
                                       (* is integer or pointer or ubknown *)
                                       (* Flags must be defined in the     *)
                                       (* definition module (components    *)
                                       (* are used in EveSyntAn) but       *)
CONST
      FlagsX = "pointer,int,none";     (* Needed for printing              *)

      noFlag = none;

      maxArrayOfASTLen = 64;          (* Maximal length of a list of ASTs  *)
```

```
      maxArrayFlagLen  = 64;          (* Maximal length of a list of Flags *)

      noCharVal = ' ';
      noCardVal = 0;

TYPE
      ArrayOfASType = ARRAY [1..maxArrayOfASTLen] OF ASType;
      ArrayOfFlags  = ARRAY [1..maxArrayFlagLen] OF Flags;


      RelationType = (eqRelTy, ltRelTy, leRelTy, gtRelTy, geRelTy, neRelTy);
                      (* This enumeration is used to select which relation *)
                      (* should be built in MakeASTRelation.               *)

      ArithmType = (minusArTy, plusArTy, orArTy,
                    timesArTy, divArTy, modArTy, andArTy);
                      (* This enumeration is used to select which arith.   *)
                      (* operation should be built in MakeASTArithm.       *)

      AtomType = (charAtom, idAtom, numberAtom);
                      (* This enumeration is used to select which leave of *)
                      (* the AST should be built in MakeASTAtom.           *)

      NumIdASType = RECORD                         (* Used for branches in  *)
                     number, name : CARDINAL;      (* Case expr.:           *)
                     ast          : ASType;        (*  number: nr of variant*)
                    END;                           (*  name: id of variant  *)
                                                   (*  ast: ast of body     *)
      OptionalASType = RECORD                     (* Used for else branch   *)
                        CASE exist : BOOLEAN OF   (* in a case expr         *)
                          TRUE : ast : ASType;    (* optional               *)
                        END;
                       END;


VAR
      emptyAST    : ASType;

(* --------------------------------------------------------------------- *)
(*                                                                       *)
(*          Procedures for creating abstract syntax trees                *)
(*                  (only used in 'EveSyntAn')                           *)
(*                                                                       *)
(* --------------------------------------------------------------------- *)

PROCEDURE MakeASTDefSeq (asts : ARRAY OF ASType; len : CARDINAL;
                         VAR x : ASType);
 (*
   Build an AST representing a DefinitionSequence out of the 'asts[0..len-1]'
   by making them to subtrees of the definition sequence.
```

```
EveAST                                                                     Page:   3


   Input: asts_0, ..., asts_(len-1)
   Output: x where
           x = defseqN
                 |
              asts_0  -  ...  -  asts_(len-1)                                 *)


PROCEDURE MakeASTDef    (idName : CARDINAL; ast : ASType;
                         flag : Flags; VAR x : ASType);
 (*
   Build an AST for a variable binding (ast may be a lambda expr -> fct def)

   Input: idName, ast
   Output: x where
           x = defN
                 |
                idN       -  ast
                 (idName)                                                     *)


PROCEDURE MakeASTLambda (idNames  : ARRAY OF CARDINAL;
                         flagList : ARRAY OF Flags;   len: CARDINAL;
                         ast : ASType; flag : Flags;
                         VAR x : ASType);
 (*
   Build an AST for a lambda abstraction (i.e. body of a fct def).
   To make the structure of a lambda asbtraction clearer, it is split
   into a lambda-head (`lheadN') containing the variables and a
   lambda-body (`lbodyN') containing the actual expression. The param
   `flag' expresses if the expression evaluates to an integer value or
   to a pointer.

   Input: idNames_0, ..., idNames_(len-1), ast
   Output: x where
           x = lambdaN
                 |
               lheadN          ----------------------        lbodyN
                 |                                             |
                idN     -     ...  -  idN                     ast
                 (idNames_0)          (idNames_(len-1))                       *)

PROCEDURE MakeASTLet    (ast1, ast2 : ASType; flag : Flags;
                         VAR x : ASType);
 (*
   Build an AST representing a local definition.
   `ast1' must contain a DefinitionSequence and `ast2' must contain an
   expression. Again, `flag' indicates wether the whole construct
   evaluates to an integer value or to a pointer.

```

```
EveAST                                                                  Page:   4

   Input: ast1, ast2
   Output: x where
           x = letN
                 |
                ast1  -  ast2                                                 *)

PROCEDURE MakeASTIf     (head, thenBody, elseBody : ASType;
                         flag : Flags; VAR x : ASType);
 (*
   Build an AST representing a conditional expression. The input asts
   are the subexpressions for head, then-branch and else-branch.

   Input: head, thenBody, elseBody
   Output: x where
           x =  ifN
                 |
                head  -  thenBody  -  elseBody                                *)

PROCEDURE MakeASTCase   (head : ASType;
                         bodies : ARRAY OF NumIdASType; len : CARDINAL;
                         else : OptionalASType;
                         flag : Flags; VAR x : ASType);
 (*
   Build an AST representing a Case Expr. Here the array `bodies'
   contains for each part all necessay computations. The AST might
   have an else branch, too.

   Input: head, bodies_0, ..., bodies_(len-1), else
   Output: x where
           x = caseN
                 |
               head  - ... -  bodyN  ------   ...   ------     [  elseN   ]
                                |                              [    |     ]
                            numberN  -  idN   -  bodies_i.ast  [ else.ast ]
                      (bodies_1.number) (bodies_i.name)

              where                i=0..len-1                                 *)

PROCEDURE MakeASTList   (cells : ARRAY OF ASType; len : CARDINAL;
                         VAR x : ASType);
 (*
   Build an AST representing a list. `cells' contains the `len' components
   of the list. The whole AST ends with a `composeN' that has the special
   node `nilN' as the second component.
   As all elements of the list must be given to this procedure via
   the parameter `cells' this procedure is only used for list constants
   (compare `MakeASTConsList').

   Input: cells_0, ..., cells_(len-1)
   Output: x where
```

```
EveAST                                                              Page:   5

           x =   composeN
                    |
                 cells_0  -  composeN
                                |
                             cells_1  - .
                                        .
                                        . -  composeN
                                                |
                                           cells_(len-1) - nilN

    Takes ASTs and combines them to a list.
    NOTICE: The last element of a list is the atomic object 'nilN',
            which represents the NIL pointer on the level of the AST.      *)

PROCEDURE MakeASTConsList   (cells : ARRAY OF ASType; len : CARDINAL;
                             ast : ASType;
                             VAR x : ASType);
 (*
   Build an AST representing a list. Differently to 'MakeASTList' here
   is also a 'ast' given that should form the end of the list. This
   'ast' will itself represent a list. Therefore, this procedure
   is used for the CONS operation.

   Input: cells_0, ..., cells_(len-1), ast (list!)
   Output: x where
           x =   composeN
                    |
                 cells_0  -  composeN
                                |
                             cells_1  - .
                                        .
                                        . -  composeN
                                                |
                                           cells_(len-1) - ast             *)

PROCEDURE MakeASTRelation (rel : RelationType; ast1, ast2 : ASType;
                           VAR x : ASType);
 (*
   Build an AST representing a relation (comparison) between two expressions.

   Input: rel, ast1, ast2      rel IN {eqRelTy,ltRelTy,leRelTy,gtRelTy,
                                       geRelTy,neRelTy}
   Output: x where
           x = rel
                |
               ast1  -  ast2                                               *)

PROCEDURE MakeASTArithm (operator : ArithmType; ast1, ast2 : ASType;
                         VAR x : ASType);
 (*
```

```
EveAST                                                                Page:   6

    Build an AST representing an arithmetical operation on two expressions.
    This includes  binary operations on INTs and BOOLs.

    Input: operator, ast1, ast2   operator IN {minusArTy,plusArTy,orArTy,
                                               timesArTy,divArTy,modArTy,
                                               andArTy}
    Output: x where
            x = operator
                   |
                 ast1  -  ast2                                                *)

PROCEDURE MakeASTSign   (ast : ASType;
                         VAR x : ASType);
 (*
    Build an AST representing a negative INT value. Hence, 'ast' must
    represent an INT value itself.

    Input: ast
    Output: x where
            x = minusSignN
                    |
                  ast                                                         *)

PROCEDURE MakeASTNot    (ast : ASType;
                         VAR x : ASType);
 (*
    Build an AST representing the negation of a BOOL value. Hence,
    'ast' must itself represent a BOOL value.

    Input: ast
    Output: x where
            x = notN
                  |
                ast                                                           *)

PROCEDURE MakeASTApplication (ast : ASType;
                              args : ARRAY OF ASType; len : CARDINAL;
                              flag : Flags; VAR x : ASType);
 (*
    Build an AST representing the application of a higher order fct to
    some argument expressions. These 'len' argument expressions are
    represented by the array 'args'.

    Input: ast, args_0, ..., args_(len-1)
    Output: x where
            x = applyN
                  |
                ast  -  args_0  -  ...  -  args_(len-1)                       *)

PROCEDURE MakeASTCall (ast : ASType;
```

```
EveAST                                                           Page:   7

                       args : ARRAY OF ASType; len : CARDINAL;
                       flag : Flags; VAR x : ASType);
 (*
   Build an AST representing a call of a first order function with the
   `len' arguments that are represented by the array `args'. Note that
   since no currying is allowed in Eve `len' must be equal to the number
   of formal parameters in the function definition of the called function.

   Input: ast, args_0, ..., args_(len-1)
   Output: x where
           x = callN
                |
               idN  -  args_0  -  ...  -  args_(len-1)                 *)

PROCEDURE MakeASTTuple  (elems : ARRAY OF ASType; len : CARDINAL;
                          VAR x : ASType);
 (*
   Build an AST representing a tuple in Eve. The `len' fields of the
   tuple will be filled by evaluating the expressions in the array `elems'.

   Input: elems_0, ..., elems_(len-1)
   Output: x where
           x = composeN
                 |
              elems_0  -  ...  -  elems_(len-1)                        *)

PROCEDURE MakeASTSelection (nr : CARDINAL; ast : ASType;
                             flag : Flags; VAR x : ASType);
 (*
   Build an AST representing the selection of one field with number `nr'
   from a tuple that is represented by `ast'.

   Input: nr, ast
   Output: x where
           x = selectN
                 |
              numberN  -  ast
                 (nr)                                                  *)

PROCEDURE MakeASTCurry  (ast : ASType; VAR x : ASType);
 (*
   Build an AST representing a curried expression. As there are no
   explicit curried expression allowed in Eve it suffices to have only
   one argument, namely the `ast' that is transformed into a curried
   expression. However, after the lambda-lifting step there might be
   also some other variables added to the this implicit curried
   expression.

   Input: ast
   Output: x where
```

```
EveAST                                                              Page:   8

           x = curryN
                |
               ast

   Remark: This procedure is needed to handle functions as first order
           objects. If a fct is used as a parameter or stored in a
           data structure ‘curryN’ is put at the root of ‘ast’ to
           denote that ‘ast’ is a higher order fct.                           *)

PROCEDURE MakeASTProgram  (ast : ASType; VAR x : ASType);
 (*
   Build an AST representing the whole program. As argument it
   takes the ‘ast’ that have been delivered from DefinitionSequence.
   The result of this procedure is then a well defined abstract syntac tree.

   Input: ast
   Output: x where
           x = programN
                |
               ast                                                            *)


PROCEDURE MakeASTAtom   (atom : AtomType;
                         charVal : CHAR; cardVal : CARDINAL;
                         flag : Flags;
                         VAR x : ASType);
 (*
   Build an AST representing a leaf in the AST. The parameter ‘arg’ tells
   which kind of leaf will be built (possible options are: a character
   constant, an integer constant and an identifier).
   If a leaf with a character value should be built ‘charVal’ contains
   the ASCII code of that value.
   If a leaf with an integer constant should be built ‘cardVal’ contains
   the actual value of the object.
   If a leaf with an identifier should be built ‘cardVal’ contains the
   name of this identifier.
   The argument flag may only be ‘pointer’ for integer values (namely
   if it is a pointer to some compound data structure.

   Input: atom, charVal, cardVal   atom IN {charAtom, idAtom, numberAtom}
   Output: x where
           x = charN              , if atom=charAtom
               (charVal)
           x = idN                , if atom=idAtom
               (cardVal)
           x = numberN            , if atom=numberAtom
               (cardVal)                                                      *)

(* ------------------------------------------------------------------------ *)
(*                                                                          *)
```

```
EveAST                                                           Page:   9

(*       Procedures that extract some information out of the AST          *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE GetIdNameAST (ast : ASType; VAR name : CARDINAL);
 (*
   This routine takes as input the AST `ast'. It checks whether this
   AST is an identifier leaf. If not an error occurs. If yes then
   the name of this identifier is extracted from the node and
   returned as output parameter `name'                                    *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*       Procedures for handling the auxiliar data structure of a         *)
(*                         ``case type table''                            *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE SetCaseType (ast : ASType; eType : EType);
 (*
   This routine checks whether `ast' is a case expression. If not an error
   occurs. If yes then this `ast' and the corresponding type `eType'
   are entered in table. This table is only needed for the translation
   of case expressions into if-then-else expressions. This translation
   is done in `Translation'.                                              *)

PROCEDURE GetCaseType (ast : ASType; VAR eType : EType);
 (*
   This procedure takes `ast' as an input and searches in the case type table
   for the corresponding `eType'. This auxiliar data structure of a case type
   table is used to speed up the translation of case expressions into
   if-then-else expressions.                                              *)

PROCEDURE DisposeCaseTypeTable;
 (*
   Discard the memory that has been allocated for the case type table.
   This table may be discarded after the execution of the translation
   in `Translate'.                                                        *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*      Procedures for getting statistical information about the AST      *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE sizeOfAST (x : ASType) : CARDINAL;
 (*
   This procedure computes the size of the AST `x' in terms of nodes in
   the AST (i.e. each node has the same weight).                          *)
```

```

PROCEDURE GetSizeOfAST (x : ASType) : LONGCARD;
 (*
   This procedure computes the size of the AST 'x' in terms of bytes.
   For that the builtin function SIZE is used to compute the size of
   a single node in the tree.                                          *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*       Procedures performing some transformations on the AST         *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE Translate (x : ASType);
 (*
   This procedure actually performs two transformations on the AST 'x':
    - All calls to predefined functions (like CONS, ADD, ...) are translated
      into the corresponding, explicit nodes (like composeN, addN, ...).
    - All case expressions are replaced by equivalent if-then-else
      expression in which the tag of the object is tested. Upon the result
      of this test the correct branch for the further computation will be
      chosen.                                                          *)

PROCEDURE AttachBoundVars (ast : ASType; VAR boundVars, freeVars : List);
 (*
   This procedure attaches to each node of the AST the set of free variables
   that occur in that subtree. Furthermore, it returns a list of
   variables that are bound in 'ast' and whose binding scopes over the
   next subtree in the AST. This is for example necessary for removing
   all variables that are bound in the DefinitionSequence part of
   a local definition from the free variables of the body expression.
   The resulting set is then the set of free variables. The local
   definition itself need not export any bound variables.
   As the top level node in the AST is a defSeqN this routine has to be
   used for attaching free variables to the nodes in the tree. A
   similar procedure 'AttachFreeVars' has the same semantics as this
   procedure but it does not return any bound variables (this is
   possible for all expressions).                                      *)

PROCEDURE Traverse (x : ASType; addedParFctsL : List; newParLL : ListList);
 (*
   The input of this procedure is the whole AST of the program ('x'),
   a list of all functions where parameters had to be added in its
   definition ('addedParFctsL') and a list of those parameter-lists that
   had to be added to the corresponding function. So, the i-th
   element of 'newParLL' contains a list of parameters that have been
   added to the function definition of the i-th element in the list
   addedParFctsL. This procedure now makes the following:

    - Each nested function definition is pruned from the AST. This can be
```

```
      done as the nested function definitions have already been put
      into a list by ‘EveSym.LiftFcts’.
    - In each function call of a function that is an element of
      ‘addedParFctsL’ the new parameters that can be fetched from
      ‘newParLL’ have to be added (for that ‘AddParam’ is used).          *)

(* ------------------------------------------------------------------------ *)
(*                                                                          *)
(*                 Procedures for lambda lifting                            *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)

PROCEDURE AddASTsToDefSeq (asts : ARRAY OF ASType; len : CARDINAL;
                           VAR ast : ASType);
 (*
   The input for this procedure is a list of lifted function definitions
   of length ‘len’ in ‘asts’. The AST ‘ast’ represents the original program
   where all local function definitions have been cut off. In this step
   all elements of ‘asts’ are put in the top level definition sequence
   before the definition of ‘ast’. The resulting ‘ast’ is then the lifted
   version of the given program.                                           *)

PROCEDURE AddParam (ast : ASType; freeVarsList : List;
                    VAR newParList : List);
 (*
   Here, ‘ast’ must be a definition of a function (otherwise an error occurs).
   ‘freeVarsList’ contains all variables that must be parameters of
   that function definition (it contains parameters of the function and
   free variables that must be changed into additional parameters of the
   function. After this procedure the function definition does not contain
   any free variables any more and can be lifted on top level.             *)

PROCEDURE GetLocalVarsFromAST (ast : ASType; VAR localVars : List);
 (*
   The input (‘ast’) must be a fcunction definition. This procedure
   extracts all local variables of the function and returns them
   in ‘localVars’. In this case local variables can only be variables
   that are defined in the DefinitionSequence of a local definition that
   is the top level construct of the ‘ast’. If the body of the function
   that is defined in ‘ast’ is no local definition ‘localVars’ will be
   the empty list.
   This procedure is only needed for the lambda lifting, namely
   in ‘EveSym.GetLists’.                                                   *)

(* ------------------------------------------------------------------------ *)
(*                                                                          *)
(*                 Procedures for code generation                           *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)
```

```
PROCEDURE getFreeVars (ast : ASType) : List;
 (*
   This function extracts the list of free variables that is associated
   to the AST 'ast'.                                                     *)

PROCEDURE getFlag (ast : ASType) : Flags;
 (*
   This function extracts the flag ('pointer' or 'int') that is associated
   to the AST 'ast'.                                                     *)

PROCEDURE DeComposeAST (ast : ASType;
                        VAR nodeKind : NodeKind;
                        VAR cardVal : CARDINAL; VAR charVal : CHAR;
                        VAR sons : ARRAY OF ASType; VAR len : CARDINAL);
 (*
   The input for this procedure is an AST 'ast'.
   This AST is split into all of its components, which are:
    - The kind of the node itself (e.g. defN, letN, ...).
    - If the node is an identifier (idN) or an integer value (numberN)
      'cardVal' contains the name of the identifier or the value of
      the integer.
    - If the node is a character (charN) value then 'charVal' contains
      the ASCII code of this character.
    - All the subtrees of the given tree 'ast' are returned in the list
      'sons' that contains 'len' entries.
   This decompostion is extensively used in code generation step
   ('EveCodeGen') as the AST has to be traversed there.                 *)

PROCEDURE GetASTFctNames (VAR fctNames, arities : List; ast : ASType);
 (*
   This procedure extracts from the AST 'ast' a list of all functions
   that are defined in that AST ('fctNames') and a list of the
   corresponding arities ('arities'). This information will be kept in
   a table locally in 'EveCode'. When code for a curried expression
   has to be generated this information is needed.                      *)

(* --------------------------------------------------------------------- *)
(*                                                                       *)
(*              Miscellaneous functions and procedures                   *)
(*                                                                       *)
(* --------------------------------------------------------------------- *)

PROCEDURE flag (eType : EType) : Flags;
 (*
   This function decides upon the structure of the given type if it is
   a compund structure (-->'pointer') or if it is an atomic structure
   (-->'int').                                                          *)

PROCEDURE PrintAST (x : ASType; nr : CARDINAL);
 (*
```

```
EveAST                                                      Page:  13

   This procedure is mainly intended for testing.
   It prints the whole AST 'x' to standard output. Different levels
   in the tree are denoted by indentation and level numbers ('nr' is
   the level number of the current AST.                               *)

END EveAST.
```

```
EveCode                                                                Page:   1

DEFINITION MODULE EveCode;   (* Data Encapsulation of Code Area Management
===========================================================================
 This module contains the opaque data type of the Code Area and
 all procedures that are necessary to write Adam assembler code into
 that area.
 ------------------------------------------------------------------------- *)

FROM FIO            IMPORT File;
FROM EveRegister    IMPORT Register;
FROM EveLabel       IMPORT Label;
FROM EveEvent       IMPORT Event;
FROM EveStringLib   IMPORT LongString, String;
FROM EveList        IMPORT List;
FROM EveAST         IMPORT ASType;


TYPE  CodeArea;                    (* Encapsulated date type of code area *)


CONST                                                  (* For printing and  *)
      CommandX = "ADD,ADDC,SUB,SUBC,SUBRC,MUL,MULC,"+ (* writing to files  *)
                 "DIV,DIVC,DIVRC,MOD,MODC,MODRC,AND,OR,"+
                 "EQUAL,EQUALC,LESS,LESSC,LESSRC,LESSEQ,LESSEQC,LESSEQRC,"+
                 "NEG,NOT,FAILED,"+
                 "COPYI,COPYP,DELP,SPLIT,SWITCH,JUMP,WAIT,CONST,"+
                 "FASTCALL,TAILCALL,CONTEXT,APPLY,SEND,SENDC,"+
                 "SUBMIT,RETURN,RETURNC,TERMINATE,"+
                 "ALLOC,READ,WRITEI,WRITEP,WRITEC,NOP,STRING,SHORT,COMMENT  ";

      ArgKindX = "register,event,label,indir,fct,constant,ref,no  ";

TYPE
      Command = (add,addC,sub,subC,subRC,mul,mulC,
                 div,divC,divRC,mod,modC,modRC,and,or,
                 equal,equalC,less,lessC,lessRC,lesseq,lesseqC,lesseqRC,
                 neg,not,failed,
                 copyI,copyP,delP,split,switch,jump,wait,const,
                 fastcall,tailcall,context, apply, send, sendC,
                 submit,return,returnC,terminate,
                 alloc,read,writeI,writeP,writeC,
                 nop,string,short,comment);
            (* ------------------------------------------------------------ *)
            (* This enumeration contains all Adam assembler instructions. *)
            (* Furthermore, it contains tags for characterizing lines     *)
            (* where only text should be generated in the resulting       *)
            (* Adam assembler file. These tags are:                       *)
            (*  nop ... creates an empty line in the output               *)
            (*  comment ... creates a line ";  " in the output            *)
            (*  short ... writes a short string (max 40 chars) to output  *)
            (*  string ... writes a long string (max 80 chars) to output  *)
```

```
EveCode                                                             Page:   2

            (* ---------------------------------------------------------- *)

      ArgumentKind = (register,event,label,indir,fct,constant,ref,no);
                   (* ------------------------------------------------- *)
                   (* The objects to which arguments of Adam instructions *)
                   (* may refer. Note:  ref is label or indir           *)
                   (* ------------------------------------------------- *)

      ArgumentKindSet = SET OF ArgumentKind;

      Argument = RECORD
                   CASE kind : ArgumentKind OF
                      register : regVal   : Register;
                    | event    : eventVal : Event;
                    | label    : labVal   : Label;
                    ELSE         value    : CARDINAL;
                   END;
                 END;

VAR   Code                    : CodeArea;   (* The generated code is here *)
      const0, const1,const2,        (* Constant values (.kind = constant) *)

      next, next2,                  (* Indirect jumps  (.kind = indir)    *)
      noArgument              : Argument;            (* Special arguments *)
      nrOfStretchedJumps,
      nrOfUnresolvedLabels    : CARDINAL;      (* Statistical information *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*                         Initializations                                *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE InitCode ;                       (* Initialization of Code Area *)

PROCEDURE GetFctNames (ast : ASType; VAR fctNames, arities : List);
 (*
   This procedure takes the whole AST as input and creates a list of all
   functions and a list of the corresponding arities. This is needed
   to store the function names (cardinals), arities together with the
   real names of the functions (strings) in a table (local data structure
   in EveCode). Hence, the symbol table can be discarded after the 2nd
   pass.
   This procedure is only called in 'EveCodeGen.CodeGen'.
   It is the only procedure that calls 'EveAST.GetASTFctNames'.          *)


(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*             Type transformations into 'Argument' type                  *)
```

```
EveCode                                                              Page:   3

(*                                                                          *)
(* ------------------------------------------------------------------------ *)

PROCEDURE RegArg (reg : Register) : Argument;
 (*
   This procedure takes a register and returns an argument of register
   type that has the value 'reg'. Necessary, because the Emit* procedures
   require parameters of 'Argument' type.                                    *)

PROCEDURE LabArg (lab : Label) : Argument;
 (*
   This procedure takes a label and returns an argument of label
   type that has the value 'lab'. Necessary, because the Emit* procedures
   require parameters of 'Argument' type.                                    *)

PROCEDURE EvArg (ev : Event) : Argument;
 (*
   This procedure takes an event and returns an argument of event
   type that has the value 'ev'. Necessary, because the Emit* procedures
   require parameters of 'Argument' type.                                    *)

(* ------------------------------------------------------------------------ *)
(*                                                                          *)
(*              Procedures writing into the code area                       *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)

PROCEDURE Emit (label : Label; command : Command;
                arg1, arg2, arg3 : Argument);
 (*
   Write one line of code into the code area (at next free place). Each
   line is split into its components, which are the input parameters.
   If a line has no label, 'label' must have the special value 'noLabel'.
   If a command has fewer than 3 arguments, the superflous arguments must
   have the special value 'noArgument'. When written into a file a line
   is produced with the components in the same order as in the parameter
   list with <TAB> characters separating the components, a ':' after the
   label and the arguments separated by ','.                                 *)

PROCEDURE EmitCall (lab : Label; whichCall : Command;
                    name : CARDINAL; arg : Argument);
 (*
   Write a line containing a fucntion call to the code area. 'whichCal' may
   only have the values 'fastcall' or 'tailcall'. In this procedure the
   table, containing function names (cardinals) and their real names
   (strings) is accessed to get the string that has to be written as
   first argument of the 'whichCall' command to the code area. Further
   procedures that needs this local table are:
    - 'EveCodeGen.EmitExport',
    - 'EveCodeGen.EmitFunction',
```

```
EveCode                                                      Page:   4

     - ‘EveCodeGen.EmitEnd’.
    The ‘arg’ parameter must specify the successor instruction.          *)

PROCEDURE EmitConstant (val : CARDINAL);
  (*
    Write a constant to the code area. ‘val’ contains a valid constant value.
    The label field gets the value ‘noLabel’, all arguments get the value
    ‘noArgument’.                                                        *)

PROCEDURE EmitString (str : String);
  (*
    Write a string to the code area. ‘str’ contains a string
    (with maximal length: ‘EveStringLib.stringLength’).
    The label field gets the value ‘noLabel’, all arguments get the value
    ‘noArgument’.                                                        *)

PROCEDURE EmitProgram;
  (*
    Write the string "PROGRAM Program"to the code area.
    The label field gets the value ‘noLabel’, all arguments get the value
    ‘noArgument’.                                                        *)

PROCEDURE EmitEndProgram;
  (*
    Write the string "END Program"to the code area.
    The label field gets the value ‘noLabel’, all arguments get the value
    ‘noArgument’.                                                        *)

PROCEDURE EmitExport (fctNames : List);
  (*
    Write an export list containing all functions in ‘fctNames’ to the
    code area. Each line of the export list starts with "EXPORT " and then
    contains the names of functions in ‘fctNames’. Each line has max.
    ‘EveCodeGen.maxExportLen’ characters. For getting the real names (strings)
    of the functions in ‘fctNames’ the local table in ‘EveCodeGen’ has to
    be accessed. The label field gets the value ‘noLabel’, all arguments
    get the value ‘noArgument’.                                          *)

PROCEDURE EmitFunction (name : CARDINAL);
  (*
    Write the head of a function definition to the code area. This head
    consists of the string "FUNCTION " and the real name of the function
    ‘name’ (access to local table of functions and their names).
    The label field gets the value ‘noLabel’, all remaining arguments get the
    value ‘noArgument’.                                                  *)

PROCEDURE EmitEnd (name : CARDINAL);
  (*
    Write the last line of a function definition to the code area. This line
    consists of the string "END " and the real name of the function
```

EveCode Page: 5

```
  'name' (access to local table of functions and their names).
  The label field gets the value 'noLabel', all arguments get the
  value 'noArgument'.                                               *)

PROCEDURE EmitFctTuple (name : CARDINAL; i : CARDINAL);
 (*
  Write a line of the form "<f,k>" to the code area. Here, 'f' is the
  real name of the function 'name' and 'k'=arity('name')+1-i. Hence,
  'i' is the number of curried arguments of the function and 'k' is
  the index of the first of these curried arguments. The translation
  from "<f,k>" to a number that contains the address of the fct and 'k'
  is done by the assembler.
  The label field gets the value 'noLabel', all arguments get the value
  'noArgument'.                                                     *)

PROCEDURE EmitComment (emptyLine : BOOLEAN);
 (*
  Write a comment line the code area. This comment line may be:
   - an empty line (denoted by the pseudo command 'nop'),
   - a line containing the comment symbol ";" and a comment string
     (denoted by the pseudo comment 'comment').
  The label field gets the value 'noLabel', all arguments get the value
  'noArgument'.                                                     *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*   Handling of the number of events, registers and labels that are     *)
(*                      used in a function.                              *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE StoreEventRegisterLabelNr (x, maxEventNr, maxRegisterNr,
                                     maxLabelNr               : CARDINAL);
 (*
  Store the numbers 'maxEventNr', 'maxRegisterNr' and 'maxLabelNr' as the
  current number of events, registers and labels, respectively, that are
  used by the the funtion 'x'. These values are stored together with
  'x' and the real name of 'x' in a local data structure.          *)

PROCEDURE GetEventRegisterLabelNr (x : CARDINAL;
                               VAR maxEventNr, maxRegisterNr,
                                   maxLabelNr               : CARDINAL);
 (*
  Fetches the number of events, registers and labels that have been used
  by the function 'x' in the output parameters 'maxEventNr', 'maxRegisterNr'
  and  'maxLabelNr'. These values must have been stored with a call to
  'StoreEventRegisterLabelNr' before. This procedure is needed in
  'AdjustCode' where for each function e.g. unresolved labels have to be
  resolved. For that the number of needed events, registers and labels
  of that function must be restored.                               *)
```

```

PROCEDURE SetEventRegisterNr (maxEventNr, maxRegisterNr : CARDINAL);
 (*
   This procedure adds a string of the form "(m,n)" to the line where
   the head of the current fct is defined. There, 'm' stands for the
   maximal number of events and 'n' for the maximal number of registers
   that are needed in that function. As these numbers are known after
   the whole code for this function has been generated they can't be
   directly set in 'EmitFunction'.                                     *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*             Marking the beginning and the end of MAIN               *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE SetMAINStart;
 (*
   This procedure marks the start of the definition of MAIN in the
   code area and remembers the index of that start in a global variable.
   This is needed to easily extract the definition of MAIN into an
   explicit file.                                                      *)

PROCEDURE SetMAINEnd;
 (*
   This procedure marks the end of the definition of MAIN in the
   code area and remembers the index of that end in a global variable.
   This is needed to easily extract the definition of MAIN into an
   explicit file.                                                      *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*               Transformations on the generated code                 *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE AdjustCode ;
 (*
   Here some transformations of the already generated code are performed.
   It is only called once from 'EveCodeGen.CodeGen' after the code
   for the program has been created. This procedure does the following:
    - It deletes all the SPLIT commands before the first function
      definition in the program (they are only necessary for nested
      Definition Sequences but not on top level).
    - It deletes the label at the first line of a function. This is an
      unresolved label and not needed as a function is called and not
      jumped to.
    - It resolves unresolved labels. After the code generation there may
      be unresolved labels (e.g. local objects that are not used in the
      body of the definition). These labels are resolved by inserting
```

```
EveCode                                                             Page:   7

           a WAIT instruction as the last line of the function, where the
           unresolved label jumps to. This creates a final signal tree.
           The number of such labels that have to be explicitly resolved
           is counted.
         - It stretches short jumps. All indirect jumps (i.e. all jumps
           except those with "JUMP label") can at most jump over
           'EveCodeGen.maxShortJump' lines. If an indirect jump is longer
           an explicit "JUMP" command has to be inserted at the next line with
           the label for the indirect jump that is too long. The indirect
           jump itself is transformed to '+1' (e.g. the following transformation
           is necessary if the label 'l1' is more than 'maxShortJump' lines away:

                         ADD r,r1,l1   ---->   ADD   r,r1,+1
                                               JUMP  l1

           The number of such stretched jumps is counted.                    *)

     (* -------------------------------------------------------------------- *)
     (*                                                                      *)
     (*         Create output and fetch information for more output          *)
     (*                                                                      *)
     (* -------------------------------------------------------------------- *)

     PROCEDURE WriteToFile;
      (*
        This procedure writes the code area into the files "<filename>.asm" and
        "<filename>.ase". The former contains all function definitions and the
        latter contains the body of MAIN as the result expression. <filename> is
        the name of the input file with which the compiler was called.
        Here the global variables for the start and the end of MAIN are needed.
        The pseudo commands are used here to determine the format of special
        lines in the output files (e.g. 'nop' for creating an empty line).     *)

     PROCEDURE StatisticsOfCode (VAR nrOfLines, nrOfExportedFcts : CARDINAL;
                                 VAR codeSize : LONGCARD);
      (*
        This procedure fetches some statistical information of the code
        generation. This contains:
         nrOfLines ... the total number of lines (including MAIN and comment
                       lines) in the generated code.
         nrOfExportedFcts ... the number of functions in the '.asm' file that are
                              included in the export list.
         codeSize ... the size of the generated code in bytes.               *)


     END EveCode.
```

```
EveEnvironment                                                      Page:   1

DEFINITION MODULE EveEnvironment;    (*  Data Encapsulation of Environment
==========================================================================
 This module contains the opaque data type of an Environment and all
 procedures that are necessary to add something to an environment,
 delete something out of an environment or extract information out of an
 environment.
 ------------------------------------------------------------------------- *)

FROM EveAST      IMPORT Flags;
FROM EveRegister IMPORT Register;
FROM EveLabel    IMPORT Label;


TYPE Environment;                    (* Hidden data type of an environment *)

CONST
     maxArrayEnvLen = 16;

TYPE
     ArrayOfEnv = ARRAY [1..maxArrayEnvLen] OF Environment;

VAR
     initEnv : Environment;    (* Empty environment (at begin of code gen. *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*             Creation and destruction of an environment                  *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE InitEnv (VAR env : Environment);
 (*
   Initialize an environment. Necessary, since 'Environment' is an
   opaque data type.                                                       *)

PROCEDURE DisposeEnv (VAR env : Environment);
 (*
   Discard the whole environment and free the allocated memory for
   the environment and for the environment descriptor (i.e. afterwards
   'env' is no environment any more).                                      *)

PROCEDURE DeleteEnv (env : Environment);
 (*
   Discard the whole environment and free the allocated memory for the
   environment. As the environment descriptor is reset but valid
   'env' is afterwards the empty environment.                              *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                Adding information to an environment                     *)
```

EveEnvironment Page: 2

```
(*                                                                       *)
(* --------------------------------------------------------------------- *)

PROCEDURE AddToEnv (VAR env : Environment; var : CARDINAL;
                    register : Register; label : Label;
                    flag : Flags);
 (*
   This procedure changes the given environment `env' by binding the
   variable `var' to a triple of `register', `label' and `flag'.
   This is done by adding one entry, containing these four values,
   to the environment `env'.                                             *)


PROCEDURE BindVar (env : Environment; var : CARDINAL; label : Label);
 (*
   This procedure changes the environment `env' by binding the variable
   `var' to the label `label'. An entry for `var' must already exist in
   `env'. In this entry the label field is replaced by `label'.          *)

(* --------------------------------------------------------------------- *)
(*                                                                       *)
(*           Extracting information from an environment                  *)
(*                                                                       *)
(* --------------------------------------------------------------------- *)

PROCEDURE lookupRegister (env : Environment; var : CARDINAL) : Register;
 (*
   Return the register to which the variable `var' is bound in the
   environment `env'. If `var' doesn't exist in `env' an error occurs.   *)

PROCEDURE lookupLabel (env : Environment; var : CARDINAL) : Label;
 (*
   Return the label to which the variable `var' is bound in the
   environment `env'. If `var' doesn't exist in `env' an error occurs.   *)

PROCEDURE lookupFlag (env : Environment; var : CARDINAL) : Flags;
 (*
   Return the flag to which the variable `var' is bound in the
   environment `env'. If `var' doesn't exist in `env' an error occurs.   *)

(* --------------------------------------------------------------------- *)
(*                                                                       *)
(*                    Duplication of an environment                      *)
(*                                                                       *)
(* --------------------------------------------------------------------- *)

PROCEDURE CopyEnv (env1 : Environment; VAR env2 : Environment);
 (*
   Copies the whole environment, with all of its entries, from `env1' to
   `env2'. This copying automatically initializes `env1' and therefore
```

```
EveEnvironment                                                     Page:   3

    there is no explicit ‘InitEnv(env2)’ necessary.                        *)

PROCEDURE FlatCopyEnv (env1 : Environment; VAR env2 : Environment);
 (*
   Creates a second reference to ‘env1’ in ‘env2’. This means that afterwards
   ‘env1’ and ‘env2’ are ALIASES, so every change in ‘env1’ also affects
   ‘env2’ and vice versa. Hence, this procedure should only be used when
   only one of these two environments will be changed afterwards.          *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*              Print an environment (only for testing)                   *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE PrintEnv (env : Environment);
 (*
   This procedure prints the whole environment. Each entry will have the
   form "(variable,register,label,flag)" in the output.
   This is only used for testing.                                          *)

END EveEnvironment.
```

```
EveEvent                                                            Page:   1

DEFINITION MODULE EveEvent;        (*  Data Encapsulation of Event Handling
===========================================================================
 This module contains the opaque data type of an Event and procedures for
 performing all necessary operations on that data structure (for creation,
 modification, destruction, ...  of events). It has the same structure as
 'EveLabel' and 'EveRegister'.
 ------------------------------------------------------------------------ *)

FROM EveStringLib  IMPORT LongString;

TYPE  Event;                                (* Hidden data type of an Event *)

CONST
      maxArrayEventLen = 64;

TYPE
      ArrayOfEvent = ARRAY [1..maxArrayEventLen] OF Event;

VAR
      noEvent : Event;                             (* Special, unique event *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                Creation and destruction of an Event                     *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE NewEvent() : Event;
 (*
   This procedure creates a new, unique event, thereby allocating
   memory for the event. Such an explicit procedure
   call is necessary as Event is an opaque data type.                     *)

PROCEDURE DropEvent (event : Event);
 (*
   This procedure discards the event 'event' and frees the memory that
   has been allocated for it. Afterwards 'event' is no valid event
   any more (i.e. no operations of this module may be performed on it).   *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                         Operations on Events                            *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE ConvertEvent (event : Event) : LongString;
 (*
   This procedure converts the event 'event' into a unique string
   that then appears in the output files. Usually, such a string begins
   with the letter "e" and then has a unique number (unique
```

```
   with respect to the function in which it is used).                  *)

PROCEDURE AssignEvent (VAR newEvent : Event; oldEvent : Event);
 (*
   This procedure copies the value of the event `oldEvent' to `newEvent'.
   The event `newEvent' need not be initialized as it is automatically
   initialized in this procedure.                                      *)

PROCEDURE equalEvents (ev1, ev2 : Event) : BOOLEAN;
 (*
   This function tests whether the two events `ev1' and `ev2' are equal.
   As `NewEvent' creates unique events, two events that have been created
   by different calls to `NewEvent' can never be equal. They are only equal
   if they have been derived via 0 or more calls to `AssignEvent' from the
   same call to `NewEvent'. Both events must already be initialized,
   otherwise an error occurs.                                          *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*  Get the state of event creation and manipulate the creation of events  *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE getMaxEventNr () : CARDINAL;
 (*
   This function returns the number of events that have been created
   since the last `ResetEvents'. Hence, this call delivers the last number
   that is already occupied by an event. This can be viewed as the state
   of the events. This function is needed to get the state of the events
   in an Eve function. This state must be stored at the end of the
   code generation of a function and it must be fetched again in
   `AdjustCode' as eventually new events have to be created there.     *)

PROCEDURE ResetEvents;
 (*
   Reset the number of already created, unique events to 1. This is done
   at the start of a function as the events only have to be unique within
   one function. Hence, in each function the creation of new events starts
   with the same value.                                                *)

PROCEDURE SetEvent (nr : CARDINAL);
 (*
   This procedure sets the number of created events to a special value.
   This is only needed in `AdjustCode' to restore the number of
   created events at the end of the code generation for a specific Eve
   function. The parameter for this procedure can therefore only be
   got by a call to `EveCode.GetEventRegisterLabelNr'.                 *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
```

```
EveEvent                                                           Page:   3

(*                 Print one event (only for testing)                      *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE PrintEvent (event : Event);
 (*
   This procedure prints one event in the same form as it will be written
   to a file (it uses 'ConvertEvent').
   This procedure is only needed for testing.                              *)

END EveEvent.
```

EveLabel   Page: 1

```
DEFINITION MODULE EveLabel;      (*  Data Encapsulation of Label Handling
===========================================================================
 This module contains the opaque data type of a Label and procedures for
 performing all necessary operations on that data structure (for creation,
 modification, destruction, ...  of labels). It has the same structure as
 `EveEvent' and `EveRegister'.
 -------------------------------------------------------------------------- *)

FROM EveStringLib  IMPORT LongString;

TYPE  Label;                                (* Hidden data type of a Label *)

CONST
      maxArrayLabelLen = 64;

TYPE
      ArrayOfLabel = ARRAY [1..maxArrayLabelLen] OF Label;

VAR
      noLabel, startLabel : Label;  (* `noLabel' creates no output in the  *)
                                    (* output file. `startLabel' is the    *)
                                    (* input label for the code generation *)
                                    (* of the whole program (it is only    *)
                                    (* used in `EveCodeGen.P').            *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                Creation and destruction of a Label                      *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE NewLabel() : Label;
 (*
   This procedure creates a new, unique label, thereby allocating
   memory for the label. Such an explicit procedure
   call is necessary as Label is an opaque data type.                      *)

PROCEDURE DropLabel (label : Label);
 (*
   This procedure discards the label `label' and frees the memory that
   has been allocated for it. Afterwards `label' is no valid label
   any more (i.e. no operations of this module may be performed on it).    *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                         Operations on Labels                            *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE ConvertLabel (label : Label) : LongString;
```

```
EveLabel                                                   Page:   2

 (*
   This procedure converts the label ‘label’ into a unique string
   that then appears in the output files. Usually, such a string begins
   with the letter "l" and then has a unique number (unique
   with respect to the function in which it is used).                  *)

PROCEDURE AssignLabel (VAR newLab : Label; oldLab : Label);
 (*
   This procedure copies the value of the label ‘oldLab’ to ‘newLab’.
   The label ‘newLab’ need not be initialized as it is automatically
   initialized in this procedure.                                      *)

PROCEDURE equalLabels (lab1, lab2 : Label) : BOOLEAN;
 (*
   This function tests whether the two labels ‘lab1’ and ‘lab2’ are equal.
   As ‘NewLabel’ creates unique labels, two labels that have been created
   by different calls to ‘NewLabel’ can never be equal. They are only equal
   if they have been derived via 0 or more calls to ‘AssignLabel’ from the
   same call to ‘NewLabel’. Both labels must already be initialized,
   otherwise an error occurs.                                          *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*  Get the state of label creation and manipulate the creation of labels  *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE getMaxLabelNr () : CARDINAL;
 (*
   This function returns the number of labels that have been created
   since the last ‘ResetLabels’. Hence, this call delivers the last number
   that is already occupied by a label. This can be viewed as the state
   of the labels. This function is needed to get the state of the labels
   in an Eve function. This state must be stored at the end of the
   code generation of a function and it must be fetched again in
   ‘AdjustCode’ as eventually new labels have to be created there.     *)

PROCEDURE ResetLabels;
 (*
   Reset the number of already created, unique labels to 1. This is done
   at the start of a function as the labels only have to be unique within
   one function. Hence, in each function the creation of new labels starts
   with the same value.                                                *)

PROCEDURE SetLabel (nr : CARDINAL);
 (*
   This procedure sets the number of created labels to a special value.
   This is only needed in ‘AdjustCode’ to restore the number of
   created labels at the end of the code generation for a specific Eve
   function. The parameter for this procedure can therefore only be
```

```
EveLabel                                                       Page:   3

     got by a call to ‘EveCode.GetEventRegisterLabelNr’.                *)

  (* ------------------------------------------------------------------ *)
  (*                                                                     *)
  (*                   Print one label (only for testing)                *)
  (*                                                                     *)
  (* ------------------------------------------------------------------ *)

  PROCEDURE PrintLabel (label : Label);
   (*
     This procedure prints one label in the same form as it will be written
     to a file (it uses ‘ConvertLabel’).
     This procedure is only needed for testing.                         *)

  END EveLabel.
```

```
EveList                                                            Page:   1

DEFINITION MODULE EveList;                (*  Data Encapsulation of a List
===========================================================================
 This module contains the opaque data type of a list of CARDINAL numbers
 and all procedures performing operations on this data type.
 -------------------------------------------------------------------------- *)

FROM EveStringLib    IMPORT LongString;

CONST
     maxListLen = 256;
     noValue    = 0;

TYPE
     ListIndex = [0..maxListLen];

     List;                        (* Hidden data type of a List of CARDINALs *)

CONST
      maxArrayListLen = 256;

TYPE
      ArrayOfList = ARRAY [1..maxArrayListLen] OF List;


VAR   emptylist : List;              (* Special, unique list. Only needed *)
                                     (* in 'EveCodeGen.P'.                *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                           Test functions on lists                       *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE existInList (x : CARDINAL; l : List) : BOOLEAN;
 (*
   Test whether the number 'x' is an element of the list 'l'.
   As with all functions the list must be initialized otherwise an
   error occurs.                                                           *)

PROCEDURE emptyList (l : List) : BOOLEAN;
 (*
   Check whether the given list 'l' is empty. Immediately after
   initialization a list is empty.                                         *)

PROCEDURE lengthOfList (l : List) : ListIndex;
 (*
   This function returns the length of the list 'l'. If the length is 0
   'emptyList' with that list must be TRUE.                                *)

(* ----------------------------------------------------------------------- *)
```

```
EveList                                                          Page:   2

(*                                                                      *)
(*                  Creation and destruction of lists                   *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE InitList (VAR l : List);
 (*
   This procedure initializes the list ‘l’. Every list must be initialized
   before it is used. This is necessary as ‘List’ is an opaque data type.
   If an uninitialized list is fed as parameter to any of the other
   procedures an error occurs.                                          *)

PROCEDURE MakeList (elems : ARRAY OF CARDINAL; len : CARDINAL;
                    VAR l : List);
 (*
   This procedure takes an array of the length ‘len’ of CARDINAL values,
   namely ‘elems’, and creates a list ‘l’ out of it that contains the
   same elements as ‘elems’ in the same order. If ‘l’ already contains
   elements they are overwritten by the new ones.                       *)

PROCEDURE DisposeList (VAR l : List);
 (*
   This procedure discards the whole list ‘l’ and frees the memory that was
   allocated for the list cells and for the list descriptor. As also
   the list descriptor is discarded ‘l’ is no list any more after a call
   of this procedure. Therefore, no operations of this module (except
   ‘InitList’) may be applied to it.                                    *)

PROCEDURE DeleteList (l : List);
 (*
   This procedure discards the whole list and frees the memory that was
   allocated for the list cells. However, contrary to ‘DisposeList’
   this procedure keeps and updates the list descriptor such that ‘l’ is
   an empty list after a call of this procedure.                        *)

(* -------------------------------------------------------------------- *)
(*                                                                      *)
(*                         Operations on lists                          *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE AddToList (x : CARDINAL; VAR l : List);
 (*
   This procedure first checks whether ‘x’ already exists in the list ‘l’.
   If yes the ‘l’ is unchanged. Otherwise the number ‘x’ is added as
   the last element to the list ‘l’.                                    *)

PROCEDURE AddToListWithoutCheckForDoubleEntry (x : CARDINAL; VAR l : List);
 (*
   This procedure adds the number ‘x’ as the last element to the list ‘l’.
```

```
EveList                                                      Page:   3

    Contrary to the previous function no check whether the number ‘x’
    already exists in ‘l’ is performed.                                  *)

PROCEDURE GetListElement (l : List; i : ListIndex) : CARDINAL;
 (*
    This function returns the ‘i’-th element of the list ‘l’. As this
    module only handles lists of CARDINALs the result value is a
    CARDINAL. If ‘l’ has fewer than ‘i’ elements an error occurs.        *)

PROCEDURE AssignList (VAR lhs : List; rhs : List);
 (*
    This procedure copies all list elements of ‘rhs’ to ‘lhs’.
    The list ‘lhs’ must already be initialized. If ‘lhs’ contains
    elements before this operation these elements are overwritten by
    the elements of ‘rhs’.                                               *)

PROCEDURE ConcatenateLists (VAR l1, l2 : List);
 (*
    This procedure combines the lists ‘l1’ and ‘l2’ to one list that
    begins with the elements ‘l1’ and then contains the elements of ‘l2’.
    The resulting list is ‘l1’. This is done by redirecting the pointer
    of the last element of ‘l1’ to the first element of ‘l2’.
    NOTE: This procedure is destructive. After a call to this procedure
          ‘l2’ and the new ‘l1’ are ALIASES i.e. the new part of ‘l1’
          and ‘l2’ are identical. Therefore, this procedure should only be
          used when only ‘l1’ but not ‘l2’ is used afterwards!           *)

PROCEDURE AppendLists (VAR l1 : List; l2 : List);
 (*
    This procedure combines the lists ‘l1’ and ‘l2’ to one list that
    begins with the elements ‘l1’ and then contains the elements of ‘l2’.
    The resulting list is ‘l1’. Contrary to the previous procedure,
    this procedure is not destructive. Instead, it copies all the
    elements of ‘l2’ and adds them one by one to ‘l1’. Furthermore,
    only those elements of ‘l2’ are added to ‘l1’ that are not
    already contained in ‘l1’. Hence, if ‘l1’ does not contain multiple
    entries of a value before it won’t contain multiple entries of a value
    afterwards.                                                          *)

PROCEDURE IntersectLists (VAR l1 : List; l2, l3 : List);
 (*
    This procedure performs a ‘‘set intersection’’ on the lists ‘l2’ and
    ‘l3’ yielding the list ‘l1’ as a result. As this operation is only
    used on lists without multiple entries the notion from set theory
    can be applied here. Hence, ‘l1’ contains those elements that occur
    in ‘l2’ and ‘l3’. The elements in ‘l1’ are in the same order as the
    elements in ‘l2’.
    NOTE: ‘l1’ need not be initialized before this procedure is called. It
          is automatically initialized within this procedure.            *)
```

```
EveList                                                        Page:   4

PROCEDURE DifferenceLists (VAR l1 : List; l2, l3 : List);
 (*
   This procedure performs a ``set difference'' on the lists `l2' and
   `l3' yielding the list `l1' as a result. As this operation is only
   used on lists without multiple entries the notion from set theory
   can be applied here. Hence, `l1' contains those elements that occur
   in `l2' but not in `l3'. The elements in `l1' are in the same order
   as the elements in `l2'.
   NOTE: `l1' need not be initialized before this procedure is called. It
         is automatically initialized within this procedure.          *)

PROCEDURE GetListIndex (l : List; name : CARDINAL) : ListIndex;
 (*
   This function searches in the list `l' for the value `name'. If
   `name' does not occur in `l' an error occurs. Otherwise, the function
   returns the position at which `name' occurs in `l' (counting starts
   with 1). Hence, the return value must be between 1 and
   lengthOfList(l).                                                   *)

PROCEDURE SetListElement (VAR l : List; i : ListIndex; x : CARDINAL);
 (*
   This procedure stores the number `x' as the `i'-th element of the
   list `l'. The value that was previously stored at that place is
   overwritten. The index `i' must be between 1 and lengthOfList(l)
   otherwise an error occurs.                                         *)

PROCEDURE RemoveFromList (VAR l1 : List; l2 : List);
 (*
   This procedure removes all numbers that are elements of `l2' from `l1'.
   The only difference to `DifferenceLists' is that the changes are
   directly performed in `l1' and no new list has to be allocated.
   As this procedure directly operates on the internal structure of a list,
   this procedure is more efficient than `DifferenceLists'.           *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*               Get statistical information about lists                  *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE GetSizeOfList (l : List) : LONGCARD;
 (*
   This function returns the size of the list `l' in bytes.
   As lists might become very long the type of the returned size is
   LONGCARD instead of CARDINAL.                                      *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*           Procedures for testing the structure of a list               *)
(*                                                                        *)
```

```
EveList                                                          Page:   5

(* ---------------------------------------------------------------------- *)

PROCEDURE PrintList (l : List);
 (*
   This procedure prints the whole list ‘l’. The output has the form:
               [l_1, l_2, ... , l_lengthOfList(l)]
   where l_1, l_2, ..., l_lengthOfList(l) are the elements of that list in
   the same order as they are stored in ‘l’.
   This procedure is only used for testing.                                *)

PROCEDURE CheckListConsistency (l : List);
 (*
   This procedure checks if the information that is stored in the list
   descriptor is consistent with the actual structure of the list itself.
   This procedure is only used for testing.                                *)

END EveList.
```

```
EveListList                                                     Page:   1

DEFINITION MODULE EveListList;   (*  Data Encapsulation of a List of Lists
=========================================================================
 This module contains the opaque data type of a list of lists of CARDINAL
 numbers (in the following called `ListList') and all procedures performing
 operations on this data type. The structure of this module is essentially
 the same as that of `EveList'. The operations in this module have
 counterparts in `EveList' with the only difference that they operate on
 lists of CARDINALs instead of CARDINALs themselves.
 ------------------------------------------------------------------------ *)

FROM EveStringLib    IMPORT LongString;
FROM EveList         IMPORT List;

CONST
     maxListListLen = 256;

TYPE
     ListListIndex = [0..maxListListLen];

     ListList;         (* Hidden data type of a List of Lists of CARDINALs *)

CONST
      maxArrayListListLen = 256;

TYPE
      ArrayOfListList = ARRAY [1..maxArrayListListLen] OF List;


VAR
      emptylistlist : ListList;               (* Special, unique ListList *)


(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*                     Test functions on ListLists                        *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE existInListList (x : List; ll : ListList) : BOOLEAN;
 (*
   Test whether the list `x' is an element of the ListList `ll'.
   As with all functions the ListList must be initialized otherwise an
   error occurs.                                                          *)

PROCEDURE emptyListList (ll : ListList) : BOOLEAN;
 (*
   Check whether the given ListList `ll' is empty. Immediately after
   initialization a ListList is empty.                                    *)

PROCEDURE lengthOfListList (ll : ListList) : ListListIndex;
```

```
EveListList                                                   Page:   2

 (*
   This function returns the length of the ListList ‘ll’. If the length is 0
   ‘emptyListList’ with that ListList must be TRUE.                          *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                Creation and destruction of ListLists                    *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE InitListList (VAR ll : ListList);
 (*
   This procedure initializes the ListList ‘ll’. Every ListList must be
   initialized before it is used. This is necessary as ‘ListList’ is an
   opaque data type. If an uninitialized ListList is fed as parameter to
   any of the other procedures an error occurs.                              *)

PROCEDURE MakeListList (elems : ARRAY OF List; len : CARDINAL;
                        VAR ll : ListList);
 (*
   This procedure takes an array of the length ‘len’ of Lists,
   namely ‘elems’, and creates a ListList ‘ll’ out of it that contains the
   same elements as ‘elems’ in the same order. If ‘ll’ already contains
   elements they are overwritten by the new ones.                            *)

PROCEDURE DisposeListList (VAR ll : ListList);
 (*
   This procedure discards the whole ListList ‘ll’ and frees the memory
   that was allocated for the ListList cells and for the ListList
   descriptor. As also the ListList descriptor is discarded ‘ll’ is no
   ListList any more after a call of this procedure. Therefore, no
   operations of this module (except ‘InitListList’) may be applied to it. *)

PROCEDURE DeleteListList (ll : ListList);
 (*
   This procedure discards the whole ListList ‘ll’ and frees the memory that
   was allocated for the ListList cells. However, contrary to
   ‘DisposeListList’ this procedure keeps and updates the ListList
   descriptor such that ‘ll’ is an empty ListList after a call of this
   procedure.                                                                *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*                       Operations on ListLists                           *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE AddToListList (x : List; VAR ll : ListList);
 (*
   This procedure first checks whether ‘x’ already exists in the ListList
```

```
EveListList                                                    Page:   3

    'll'. If yes the ListList 'll' is unchanged. Otherwise the list 'x' is
    added as the last element to the ListList 'll'.                       *)

PROCEDURE GetListListElement (ll : ListList; i : ListListIndex) : List;
 (*
    This function returns the 'i'-th element of the ListList 'll'. As this
    module only handles Lists of Lists of CARDINALs the result value is a
    List of CARDINALs. If 'll' has fewer than 'i' elements an error occurs. *)

PROCEDURE AssignListList (VAR lhs : ListList; rhs : ListList);
 (*
    This procedure copies all ListList elements of 'rhs' to 'lhs'.
    The ListList 'lhs' must already be initialized. If 'lhs' contains
    elements before this operation these elements are overwritten by
    the elements of 'rhs'.                                                 *)

PROCEDURE ConcatenateListLists (VAR ll1 : ListList; ll2 : ListList);
 (*
    This procedure combines the ListLists 'll1' and 'll2' to one ListList
    that begins with the elements 'll1' and then contains the elements of
    'll2'. The resulting ListList is 'll1'. This is done by redirecting the
    pointer of the last element of 'll1' to the first element of 'll2'.
    NOTE: This procedure is destructive. After a call to this procedure
          'll2' and the new 'll1' are ALIASES i.e. the new part of 'll1'
          and 'll2' are identical. Therefore, this procedure should only be
          used when only 'll1' but not 'll2' is used afterwards!           *)

PROCEDURE AppendListLists (VAR ll1 : ListList; ll2 : ListList);
 (*
    This procedure combines the ListLists 'll1' and 'll2' to one ListList that
    begins with the elements 'll1' and then contains the elements of 'll2'.
    The resulting ListList is 'll1'. Contrary to the previous procedure,
    this procedure is not destructive. Instead, it copies all the
    elements of 'll2' and adds them one by one to 'll1'. Furthermore,
    only those elements of 'll2' are added to 'll1' that are not
    already contained in 'll1'. Hence, if 'll1' does not contain multiple
    entries of a value before it won't contain multiple entries of a value
    afterwards.                                                            *)

PROCEDURE GetListListIndex (ll : ListList; x : List) : ListListIndex;
 (*
    This function searches in the ListList 'll' for the list 'x'. If
    'x' does not occur in 'll' an error occurs. Otherwise, the function
    returns the position at which 'x' occurs in 'll' (counting starts
    with 1). Hence, the return value must be between 1 and
    lengthOfListList(ll).                                                  *)

PROCEDURE SetListListElement(VAR ll : ListList; i : ListListIndex; x : List);
 (*
    This procedure stores the list 'x' as the 'i'-th element of the
```

```
EveListList                                                    Page:   4

   ListList ‘ll’. The value that was previously stored at that place is
   overwritten. The index ‘i’ must be between 1 and lengthOfListList(ll)
   otherwise an error occurs.                                              *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*        Procedure for printing a List of Lists (only for testing)       *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE PrintListList (ll : ListList);
 (*
   This procedure prints the whole ListList ‘ll’. The output has the form:
                [ll_1, ll_2, ... , ll_lengthOfListList(ll)]
   where ll_1, ll_2, ..., ll_lengthOfListList(ll) are the elements of that
   ListList in the same order as they are stored in ‘ll’.
   This procedure is only used for testing.                               *)

END EveListList.
```

```
EveRegister                                                        Page:   1

DEFINITION MODULE EveRegister;  (*  Data Encapsulation of Register Handling
===========================================================================
 This module contains the opaque data type of a Register and procedures for
 performing all necessary operations on that data structure (for creation,
 modification, destruction, ...  of registers). It has the same structure
 as ‘EveEvent’ and ‘EveLabel’.
 ------------------------------------------------------------------------ *)

FROM EveStringLib  IMPORT LongString;

TYPE  Register;                          (* Hidden data type of a Register *)

CONST
      maxArrayRegisterLen = 64;

TYPE
      ArrayOfRegister = ARRAY [1..maxArrayRegisterLen] OF Register;

VAR
      noRegister : Register;                   (* Special, unique register *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*                Creation and destruction of a Register                  *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE NewRegister() : Register;
 (*
   This procedure creates a new, unique register, thereby allocating
   memory for the register. Such an explicit procedure
   call is necessary as Register is an opaque data type.                  *)

PROCEDURE DropRegister (register : Register);
 (*
   This procedure discards the register ‘register’ and frees the memory that
   has been allocated for it. Afterwards ‘register’ is no valid register
   any more (i.e. no operations of this module may be performed on it).   *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*                       Operations on Registers                          *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE ConvertRegister (reg : Register) : LongString;
 (*
   This procedure converts the register ‘reg’ into a unique string
   that then appears in the output files. Usually, such a string begins
   with the letter "r" and then has a unique number (unique
```

EveRegister Page: 2

```
   with respect to the function in which it is used).                 *)

PROCEDURE AssignRegister (VAR newReg : Register; oldReg : Register);
 (*
   This procedure copies the value of the register ‘oldReg’ to ‘newReg’.
   The register ‘newReg’ need not be initialized as it is automatically
   initialized in this procedure.                                     *)

PROCEDURE equalRegisters (reg1, reg2 : Register) : BOOLEAN;
 (*
   This function tests whether the two registers ‘reg1’ and ‘reg2’ are equal.
   As ‘NewRegister’ creates unique registers, two registers that have been
   created by different calls to ‘NewRegister’ can never be equal. They are
   only equal if they have been derived via 0 or more calls to
   ‘AssignRegister’ from the same call to ‘NewRegister’. Both registers must
   already be initialized, otherwise an error occurs.                 *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*              Get the state of register creation and                    *)
(*               manipulate the creation of registers                     *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE getMaxRegisterNr () : CARDINAL;
 (*
   This function returns the number of registers that have been created
   since the last ‘ResetRegisters’. Hence, this call delivers the last number
   that is already occupied by an register. This can be viewed as the state
   of the registers. This function is needed to get the state of the
   registers in an Eve function. This state must be stored at the end of the
   code generation of a function and it must be fetched again in
   ‘AdjustCode’ as eventually new registers have to be created there.  *)

PROCEDURE ResetRegisters;
 (*
   Reset the number of already created, unique registers to 1. This is done
   at the start of a function as the registers only have to be unique within
   one function. Hence, in each function the creation of new registers starts
   with the same value.                                               *)

PROCEDURE SetRegister (nr : CARDINAL);
 (*
   This procedure sets the number of created registers to a special value.
   This is only needed in ‘AdjustCode’ to restore the number of
   created registers at the end of the code generation for a specific Eve
   function. The parameter for this procedure can therefore only be
   got by a call to ‘EveCode.GetRegisterRegisterLabelNr’.             *)

(* ---------------------------------------------------------------------- *)
```

```
EveRegister                                                      Page:   3

(*                                                                      *)
(*                  Print one register (only for testing)               *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE PrintRegister (register : Register);
 (*
   This procedure prints one register in the same form as it will be written
   to a file (it uses 'ConvertRegister').
   This procedure is only needed for testing.                           *)

END EveRegister.
```

```
EveStringLib                                                      Page:   1

DEFINITION MODULE EveStringLib;          (* Encapsulating the type String
==========================================================================
 This module contains the data type of a string (in versions with different
 lengths) and all procedures for performing operations on this data type.
 ------------------------------------------------------------------------ *)

CONST
      stringEnd       = 0C;                    (* End of string character *)

      stringLength      = 40;           (* Length of a normal string    *)
      longStringLength  = 80;           (* Length of a long string      *)
      xlongStringLength  = 900;         (* Length of a extra long string *)

TYPE
    String       = ARRAY [0..stringLength-1]       OF CHAR; (*Normal String*)
    LongString   = ARRAY [0..longStringLength-1]   OF CHAR; (*Long String  *)
    XLongString  = ARRAY [0..xlongStringLength-1]  OF CHAR; (*XLong String *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*             Some important operations on strings                        *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE length (line : ARRAY OF CHAR) : CARDINAL;
 (*
   Determine the actual length of a string.                               *)

PROCEDURE Slice (VAR result : ARRAY OF CHAR;
                 VAR source : ARRAY OF CHAR;
                     position, length : CARDINAL);
 (*
   Get the substring of 'source' that starts at the index 'position' and
   is 'length' characters long. The resulting string is stored in 'result'.
   If the string 'source' is not long enough 'reult' is truncated
   appropriately.                                                         *)

PROCEDURE equalStr (str1, str2 : ARRAY OF CHAR) : BOOLEAN;
 (*
   Test whether the two strings 'str1' and 'str2' contain the same
   characters in the same order.                                          *)

PROCEDURE CardToStr (    card : CARDINAL;
                     VAR str  : ARRAY OF CHAR;
                     VAR ok   : BOOLEAN);
 (*
   Convert a CARDINAL number, namely 'card', to a string 'str' that
   represents the value 'str'. Iff the conversion was successful 'ok'
   returns TRUE.                                                          *)
```

```
EveStringLib                                              Page:   2

PROCEDURE Append (VAR str1 : ARRAY OF CHAR; str2 : ARRAY OF CHAR);
 (*
   Add the characters of 'str2' at the end of 'str1'.                    *)

PROCEDURE AddChar (ch : CHAR; VAR str : ARRAY OF CHAR);
 (*
   Add one character, namely 'ch', at the end of 'str'.                  *)

PROCEDURE DeleteLastChar (ch : CHAR; VAR str : ARRAY OF CHAR;
                          VAR ok : BOOLEAN);
 (*
   This procedure gets the last character of the string 'str' that is not
   a blank. If this character is not 'ch' then 'ok' is set to FALSE and no
   deletion of a character takes place. Otherwise, this character is
   deleted by replacing it by a 0C character and 'ok' is set to TRUE.     *)

PROCEDURE LongStringToString (longStr : LongString) : String;
 (*
   This procedure converts a LongString to a String. If the actual
   length of 'longStr' is bigger than 'stringLength' then the rest of
   the LongString is truncated.                                           *)

PROCEDURE CutOffFromChar (ch : CHAR; VAR str : ARRAY OF CHAR);
 (*
   This procedure deletes all characters after (and including) the first
   'ch' character in the String 'str'. This is done by replacing the
   first 'ch' character by a 0C character.                                *)

END EveStringLib.
```

```
EveSym                                                              Page:   1

DEFINITION MODULE EveSym; (* Data Encapsulation of Symbol Table & Symbol List
=============================================================================
 This module contains the two opaque data types of a symbol table and of a
 symbol list, where the later is essentially a list of pointers to the
 former. Hence, the important information of all objects in an Eve program
 is stored in the symbol table. The symbol list is only used to handle
 the scoping of the Eve objects correctly (Symbol list is organized in a
 stack like fashion. New entries are added to the list when  a new block
 in the Eve program is entered and the local elements of that block are
 deleted from the symbol list at the end of that block).
 Furthermore, this module contains all procedures that operate on these
 two data types.
 -------------------------------------------------------------------------- *)

FROM EveType     IMPORT EType;
FROM EveAST      IMPORT ASType;
FROM EveList     IMPORT List;
FROM EveListList IMPORT ListList;

CONST
     noObject = 0;                                 (* Special unique object *)
     OpKindX = "parKind,fctKind,constKind,typeKind,errorKind";
                                                   (* Only for printing     *)
TYPE
     Object = CARDINAL;                           (* Index in symbol table  *)
     OpType = EType;                              (* Operand Types is EType *)
     OpKind = (parKind, fctKind, constKind, typeKind, errorKind);

     Operand = RECORD                            (* Operand gives the       *)
                 name    : CARDINAL;             (* description of any      *)
                 CASE  kind : OpKind  OF         (* object in Eve (function *)
                   fctKind :   freeVars  : List; (* variable, parameter,..) *)
                               occurIn   : List;
                 END;
                 type    : OpType;               (* Type of the object      *)
                 level   : CARDINAL;             (* Static nesting level    *)
                 forward : BOOLEAN;              (* Object defined forward? *)
                 ast     : ASType;               (* AST for eval. object    *)
               END;

VAR
      objOfMAIN          : Object;        (* Index of MAIN in symbol table *)
      opOfMAIN           : Operand;       (* Operand description of MAIN   *)
      nrPredef           : CARDINAL;      (* Nr. of predef objects         *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*      Procedures for adding Eve objects to the symbol table             *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)
```

```
EveSym                                                          Page:   2


  (*
    The following procedures add a new entry to the symbol table and add the
    index of this new entry to the symbol list. If there is already an
    entry on the same level that has ‘name’ in the name field of the operand
    then an error occurs. The procedure below only differ in the value
    that is written to the ‘kind’ field of the operand.
    The default entries for the other fields are:
      name   ... ‘name’ (parameter of the procedure)
      type   ... ‘eType’ (parameter of the procedure; this field may be
                 manipulated by the following procedures:
                   AssignType  ... explicitly setting the type of an object
                   MakeForward*, ResolveForward* ... if there is a forward
                      definition of that onject that only defines its
                      type but not its value.
      level  ... CurrentLevel() (the static nesting level of the current
                 block in the Eve program)
      forward ... FALSE (only changed by MakeForward*, ResolveForward* )
      ast     ... emptyAST (this field may later be filled by a call to
                  ‘AssignAST’).                                         *)

PROCEDURE NewConst (name : CARDINAL; eType : EType);
 (*
    Add constant operand to symbol table and symbol list
    (i.e. .kind := constKind).                                          *)

PROCEDURE NewType (name : CARDINAL);
 (*
    Add type operand to symbol table and symbol list
    (i.e. .kind := typeKind). The ‘.type’ field will later be explicitly
    set via a call to ‘AssignType’.                                     *)

PROCEDURE NewPar (name : CARDINAL; eType : EType);
 (*
    Add parameter operand to symbol table and symbol list
    (i.e. .kind := parKind).                                            *)

PROCEDURE NewFct (name : CARDINAL; eType : EType);
 (*
    Add function operand to symbol table and symbol list
    (i.e. .kind := fctKind).  The fields ‘.freeVars’, containing all free
    variables that occur in the body of the function definition, and
    ‘.occurIn’, containing all functions that are called in the body
    of the function definition, are initialized in this procedure.      *)

PROCEDURE NewParList (name : CARDINAL; idList : ARRAY OF CARDINAL;
                      typeList : ARRAY OF EType; len : CARDINAL  );
 (*
    Add ‘len’ parameter operands to the symbol table and symbol list
    (i.e. ‘.kind’ := parKind). If ‘name’ does not exist or is not
```

```
EveSym                                                      Page:   3

    a function an error occurs. In this procedure 'NewPar' is called
    with 'idList[i]' and 'typeList[i]' for each 1<=i<=len.          *)

PROCEDURE NewIdent (VAR name : CARDINAL; eType : EType; ast : ASType);
 (*
    Add a new constant operand to the symbol table but NOT to the symbol
    list. This procedure gets a new, unique identifier by a call to
    'EveLexAn.GetNewIdent' and uses it as the '.name' field of the new
    entry in the symbol table. This procedure is needed in
    'EveAST.EliminateCASE' to get variables where the tag and the body
    of a union can be bound to.                                     *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*      Searching for entries in the symbol list and extracting        *)
(*    information from the corresponding entry in the symbol table     *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE SearchType (name : CARDINAL; VAR eType : EType);
 (*
    Search in the symbol list for an operand of 'typeKind' with the 'name'
    in the '.name' field. If such an operand exists 'eType' gets the
    type of that operand. Otherwise a semantic error occurs.        *)

PROCEDURE GetType (name : CARDINAL; VAR eType : EType);
 (*
    Same as 'SearchType' but if no appropriate operand is found in the
    symbol list a compiler error occurs.                            *)

PROCEDURE FindObject (name : CARDINAL; VAR obj : CARDINAL; VAR op : Operand);
 (*
    Search in the symbol list for an operand that has 'name' in the '.name'
    field. If such an operand is found, 'obj' gets the index of this
    operand in the symbol table and 'op' gets the contents of the
    operand itself. If it is not found then 'obj' gets the special value
    'noObject' and 'op' is undefined in this case.                  *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*           Explicit setting of fields of certain operands            *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE AssignType (name : CARDINAL; eType : EType);
 (*
    Search in the symbol list for an operand that has 'name' in the '.name'
    field. If such an operand is found, and its type is 'errorType'
    then this '.type' field is set to 'eType'. Otherwise an error occurs.  *)
```

```
EveSym                                                               Page:   4

PROCEDURE AssignAST  (name : CARDINAL; ast : ASType);
 (*
   Search in the symbol list for an operand that has ‘name’ in the ‘.name’
   field. If such an operand is found, and its AST is ‘emptyAST’
   then this ‘.ast’ field is set to ‘ast’. Otherwise an error occurs.     *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*                  Handling of forward definitions                    *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

 (* -------------------------------------------------------------------
   All the MakeForward* procedures search for an operand with ‘name’ in
   the ‘.name’ field (error if not found). Hence, even in a forward
   definition the Eve object first has to be entered to the symbol list
   with one of the ‘New*’ functions before a ‘MakeForward*’ can be called.
   In the found operand the ‘.forward’ field is set to TRUE and the
   ‘.type’ field is set to ‘eType’ (parameter of the function).
   In the following cases errors occur:
    - ‘name’ isn’t found in the symbol list.
    - The level of the found operand is different from the current level
      (i.e. there is no operand with the name ‘name’ on this level).
    - The found operand has a wrong ‘.kind field (the ‘.kind’ field
      in e.g. ‘MakeForwardType’ must have the value ‘typeKind’).
   A compiler restriction occurs if:
    - The same forward definition of an Eve object occurs twice on the
      same level (if different forward definitions of the same Eve object
      occur on the same level a semantic error will be produced).
   ------------------------------------------------------------------- *)

PROCEDURE MakeForwardType (name : CARDINAL; eType : EType);
 (*
   Changes the type operand of ‘name’ into a forward operand and sets its
   ‘.type’ field to ‘eType’. Checks see above.                            *)

PROCEDURE MakeForwardConstant (name : CARDINAL);
 (*
   Changes the constant operand of ‘name’ into a forward operand.
   Checks see above.                                                      *)

PROCEDURE MakeForwardFct (name : CARDINAL);
 (*
   Changes the function operand of ‘name’ into a forward operand.
   Checks see above.                                                      *)

 (* -------------------------------------------------------------------
   All the ResolveForward* procedures search for an operand with ‘name’ in
   the ‘.name’ field (error if not found). In the found operand the
   ‘.forward’ field is set to FALSE. If an ‘ast’ parameter is submitted
```

```
EveSym                                                      Page:   5

    to the procedure the `.ast' field of the found operand is set to this
    `ast'.
    In the following cases errors occur:
     - `name' isn't found in the symbol list.
     - The `.forward' field of the found operand is FALSE.
     - The found operand has a wrong `.kind field (the `.kind' field
       in e.g. `ResolveForwardType' must have the value `typeKind').
     - An `ast' is submitted and the `.ast' field of the found operand has
       not the value `emptyAST'.
  ------------------------------------------------------------------------ *)

PROCEDURE ResolveForwardType (name : CARDINAL);
 (*
    Changes the type operand of `name' from a forward operand back to a
    normal operand. Checks see above.                                    *)

PROCEDURE ResolveForwardConstant (name : CARDINAL; ast : ASType);
 (*
    Changes the constant operand of `name' from a forward operand back to a
    normal operand. Thereby, the `.ast' field of the operand is set
    to `ast'. Checks see above.                                          *)

PROCEDURE ResolveForwardFct (name : CARDINAL; ast : ASType);
 (*
    Changes the function operand of `name' from a forward operand back to a
    normal operand. Thereby, the `.ast' field of the operand is set
    to `ast'. Checks see above.                                          *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*       Get information about functions from the symbol table            *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE ExtractParTypes(name : CARDINAL;
                          VAR typeList : ARRAY OF EType; VAR len : CARDINAL);
 (*
    Search in the symbol list for an operand that has `name' in the `.name'
    field. If such an operand is not found or is not a function then an
    error occurs. Otherwise `typeList' gets the types of the parameters
    (excluding the type of the return value) of the Eve function `name'.
    `len' returns the number of the parameters of `name'.
    This information is actually got via a call to `EveType.GetParTypeList'.*)

PROCEDURE ExtractParNames (name : CARDINAL;
                           VAR idList : ARRAY OF CARDINAL;
                           VAR len : CARDINAL);
 (*
    Search in the symbol list for an operand that has `name' in the `.name'
    field. If such an operand is not found or is not a function then an
```

```
EveSym                                                            Page:   6

    error occurs. Otherwise ‘idList’ gets the names of the parameters of
    the Eve function ‘name’. ‘len’ returns the number of the parameters of
    ‘name’. This information is actually got via a call to
    ‘EveType.GetParTypeList’.                                             *)

PROCEDURE arity (name : CARDINAL) : CARDINAL;
 (*
    Search in the symbol list for an operand that has ‘name’ in the ‘.name’
    field. If such an operand is not found or is not a function then an
    error occurs. Otherwise the arity of the function is returned.        *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*      Handling of blocks in Eve (includes scoping of variables)        *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE BeginFct (name : CARDINAL);
 (*
    This procedure opens a new block for a function and saves the state of
    the current block.

    Search in the symbol list for an operand that has ‘name’ in the ‘.name’
    field. If such an operand is not found or is not a function then an
    error occurs. Otherwise the index of the the current procedure and the
    current state of the symbol list are pushed on a stack.
    The new current procedure is the function ‘name’ and the counter of
    the nesting levels is increased by one.                               *)

PROCEDURE EndFct (name : CARDINAL);
 (*
    This procedure closes the current block for a function and restores the
    state of the surrounding block.

    Get the topmost two entries of the stack. They contain
      1) the previous state of the symbol list of the surrounding block and
      2) the current procedure of the surrounding block.
    Furthermore, the counter of the nesting levels is decreased by one.   *)

PROCEDURE BeginBlock;
 (*
    This procedure opens a new block and saves the state of the current block.

    The index of the current procedure and the current state of the symbol
    list are pushed on the stack. The counter of the nesting levels is
    increased by one.                                                     *)

PROCEDURE EndBlock ;
 (*
    This procedure closes the current block and restores the state of the
```

```
EveSym                                                                Page:   7

   surrounding block.

   Get the topmost two entries of the stack. They contain
     1) the previous state of the symbol list of the surrounding block and
     2) the current procedure of the surrounding block.
   Furthermore, the counter of the nesting levels is decreased by one.      *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*              Miscellaneous functions and procedures                    *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE isParOrConst (name : CARDINAL) : BOOLEAN;
 (*
   Search in the symbol list for an operand that has `name' in the `.name'
   field. If such an operand is not found an error occurs.
   Otherwise the function returns TRUE iff the `.kind' field has the
   value `parKind' or `fctKind'. This function is necessary to decide in
   `EveSyntAn' whether an Eve object is higher order or first order.      *)

PROCEDURE DeleteVar (name : CARDINAL);
 (*
   If the last entry in the symbol list has `name' in the `.name' field this
   entry is deleted from the symbol list. Otherwise an error occurs.
   This procedure is needed in an Eve CASE expression where in the different
   branches the value of the union is bound to new identifiers that only
   scope over that branch and can be deleted afterwards.                  *)

PROCEDURE CurrentLevel () : CARDINAL;
 (*
   This function returns the current static nesting level of the Eve program.
   This value is kept in a global variable of this module.                *)

PROCEDURE TestAdd (name : CARDINAL);
 (*
   Search in the symbol list for an operand that has `name' in the `.name'
   field. If such an operand is not found an error occurs.
   If `name' was defined in a surrounding block then it is global here and
   it is added to the `.freeVars'  field of the current function if it
   is a variable or it is added to the `.occurIn' field of the current
   function if it is a function. This is the only procedure that creates
   entries in the fields `.freeVars' and `.occurIn'. the information of
   these fields is needed for the lambda lifting step.                    *)

PROCEDURE existInSymbolTable (name : CARDINAL) : BOOLEAN;
 (*
   Search in the symbol table for an operand that has `name' in the `.name'
   field. This function returns TRUE iff such an operand exists and this
   operand is not a forward operand (i.e. the `.forward' field must
```

```
EveSym                                                         Page:   8

   have the value FALSE).                                             *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*            Explicit operations on the special constant MAIN        *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE SetMAINFctType (eType : EType; obj : Object; VAR op : Operand);
 (*
   This procedure first checks whether `obj' is the indes of MAIN in the
   symbol table (if not then an error occurs). Then the operand of
   MAIN in the symbol table and the opperand `op' get `eType' as the `.type'
   field. The operand `op' shall contain the operand of MAIN before such
   that it contains the operand of MAIN with the correct type
   afterwards. However, this is not checked in that procedure.          *)

PROCEDURE GetTypeOfMain (VAR found : BOOLEAN; VAR eType : EType);
 (*
   This procedure sets `found' to TRUE iff in the operand for MAIN a type
   has been entered (`.type' field). In any case `eType' gets the type
   of MAIN (if no type has been entered yet `eType' will be `errorType').  *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*             Procedures needed for the lambda lifting                *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE GetLists (VAR nameList : List;
                    VAR freeVarsLList, occurInLList, exceptLList : ListList);
 (*
   This procedure goes through the whole symbol table and if it finds a
   function it adds its name to `nameList'. Furthermore, the list of free
   variables of that function (`.freeVars' field) is added to `freeVarsLList'
   and the list of functions that are called in the current function
   (`.occurIn' field) is added to `occurInLList'. The list of all
   variables that are defined locally in the current function is added
   to `exceptLList' (this list is got by a call to
   `EveAST.GetLocalVarsFromAST'). Note that in the output parameters the
   i-th component of `nameList' corresponds with the i-th components of
   `freeVarsLList', `occurInLList' and `exceptLList'.                   *)

PROCEDURE LiftFcts (ast : ASType; nameList : List; fVLL, oILL : ListList;
                    VAR addedParFctsL : List; VAR newParLL : ListList;
                    VAR asts : ARRAY OF ASType; VAR len : CARDINAL);
 (*
   This procedure is the main part of the lambda lifting step (it is called
   from `EveLambdaLift.LambdaLift'). It goes through the symbol table
   (omitting the predefined functions) and puts all nested functions in an
```

```
EveSym                                                       Page:   9

    array (the names of these functions are put into ‘addedParFctsL’ and the
    ASTs of these function definitions are put into ‘asts’). Thereby, add
    the lifted variables to the definitions of
    these functions (the i-th component of ‘fVLL’ contains a list of
    all parameters that must occur in the definition of the function
    that is the i-th component of ‘nameList’). Those variables that have
    not been parameters of the function so far and are added in this
    step are put into the ListList ‘newParLL’. ‘len’ is length of the
    resulting lists (and lists of lists).                                *)

(* ------------------------------------------------------------------ *)
(*                                                                    *)
(*   Explicit deallocation of memory for symbol list and symbol table *)
(*                                                                    *)
(* ------------------------------------------------------------------ *)

PROCEDURE DisposeTypes ;
 (*
    This procedure discards all types that are stored in the symbol table.
    For that the symbol table is traversed. As the types are only needed in
    the first two passes this procedure may be called immediately before
    the code generation.                                                 *)

PROCEDURE DisposeSymbolList ;
 (*
    This procedure discards the whole symbol list. As the symbol list is
    only needed for handling the scoping of variables correctly, it can
    be discarded after the first pass.                                   *)

PROCEDURE DisposeSymbolTable;
 (*
    This procedure discards the whole symbol table. The symbol table is
    needed only for the first two passes. The parts of information that
    are also needed in the code generation are extracted from the symbol
    table with some procedures like ‘EveCode.GetFctNames’. Hence, this
    procedure may be called after a call to ‘EveCode.GetFctNames’
    immediately before the code generation.                              *)

(* ------------------------------------------------------------------ *)
(*                                                                    *)
(*          Get statistical information about the symbol table        *)
(*                                                                    *)
(* ------------------------------------------------------------------ *)

PROCEDURE StatisticsOfSymTable (VAR size : CARDINAL;
                                VAR sizePercent : CARDINAL);
 (*
    This procedure gets in ‘size’ the size of the symbol table in terms of
    entries. In ‘sizePercent’ the usage of the whole symbol table in
    percent is returned (again in terms of entries).                     *)
```

```
EveSym                                                          Page:  10


(* ------------------------------------------------------------------------ *)
(*                                                                          *)
(*             Some print operations (only for testing)                     *)
(*                                                                          *)
(* ------------------------------------------------------------------------ *)

PROCEDURE PrintOp (op : Operand);
 (*
   Print all information of one operand 'op'. This includes:
    - name (the number as well as the real name),
    - kind
    - if it is a function:
       + the list of free variables ('freeVars')
       + the list of functions that are called in the function ('occurIn')
    - type
    - level
    - forward
    - information whether the '.ast' field is empty or not.                *)

PROCEDURE PrintDependencies;
 (*
   Print a table of all dependencies of all functions in the symbol table.
   For that the symbol table has to be traversed. For each function in the
   symbol table the contents of the '.freeVars' and '.occurIn' fields
   are printed (the real names not the numbers are printed!).              *)

PROCEDURE PrintTable (all : BOOLEAN);
 (*
   Print the whole symbol table. If all is TRUE also the predefined objects
   are printed otherwise they are omitted. In the printing the lists
   '.freeVars' and '.occurIn' are omittet. The '.ast' field is also
   omitted.                                                                *)

PROCEDURE PrintList  (all : BOOLEAN);
 (*
   Print the whole symbol list. Here only the names (real names as well as
   numbers) of the operands that are in the symbol list are printed.       *)

END EveSym.
```

```
EveType                                                          Page:   1

DEFINITION MODULE EveType;                (*  Data Encapsulation of Types
==========================================================================
 This module contains the opaque data type of an Eve type and all
 procedures that are necessary to operate on this data type.
 ------------------------------------------------------------------------- *)

FROM FIO   IMPORT File;

TYPE
      EType;                           (* Hidden data type of an Eve type *)

CONST
     dummyName = 1;

TYPE
     NamedType = RECORD                        (* A name with the Eve type *)
                   name : CARDINAL;            (* that is bound to it      *)
                   type : EType;               (* (needed for tuples and   *)
                 END;                          (* unions in Eve)           *)

     FwType = RECORD                           (* Tells whether a type has *)
               CASE isForward : BOOLEAN OF     (* been defined via a       *)
                 TRUE : type : EType;          (* definition. Yes ==>      *)
               END;                            (* 'type' contains the type *)
              END;

CONST
     maxParNr  = 8;                            (* Max number of parameters *)
                                               (* of a function            *)
                                               (* --> Compiler restriction *)
VAR
     errorType, anyType, anyListType,          (* Predefined types i.e.    *)
     boolType, charType, intType, strType,     (* that are needed by       *)
  (* FunctionTypes for predef. fcts *)         (* predefined objects and   *)
     iiiType, iiType, biType,                  (* therefore already given  *)
     bbbType, bbType,                          (* from the beginning.      *)
     ccType, ciType, icType,
     bccType, biiType, bssType,
     aLaaLType, aaLType, aLaLType, baLType,
     baType                                      : EType;

     noForward                                   : FwType;

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*         Procedures for creating and destructing Eve types              *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE CreateType (VAR x : EType);
```

```
EveType                                                              Page:   2

 (*
   This procedure initializes the type ‘x’. Before a type is used it must
   be created with a call of this procedure. This is necessary as ‘EType’
   is an opaque data type.                                                 *)

PROCEDURE DisposeType (VAR x : EType);
 (*
   This procedure discards the type ‘x’ and frees the memory that has been
   allocated for it. As the types are only needed in the first two passes
   they may be deleted immediately before the code generation (see
   ‘EveSym.DisposeTypes’ that discards all existing types).                *)

PROCEDURE DisposeTypeTab;
 (*
   This procedure explicitly deletes the local data structure of a type
   table that is only needed for the creation of a type description file.  *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*          Procedures for making compound Eve types                      *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

 (* ----------------------------------------------------------------------
    All Make*Type procedures take one or more types (most of them named
    types) and create a compound type by building a tuple, union, ...
    out of it. However, the type may already be defined as a forward to
    allow indirectly recursive definitions. In this case the type has
    already been created but is empty. As other types may refer to
    this so far empty type it has to be used in the creation of the
    new compound type. Therefore, each procedure has the parameter ‘fw’
    telling if the type has been defined as forward type and if yes it
    contains the empty type. If it is not a forward type a new type
    has to be created. In any case ‘x’ contains afterwards the new
    compound type.
    ---------------------------------------------------------------------- *)

PROCEDURE MakeListType (VAR x : EType; y : EType; fw : FwType);
 (*
   Take the type ‘y’ and create a list type where each list element is
   of type ‘y’. If it has been defined as forward type ‘fw.forward’=TRUE
   and ‘fw.type’ contains the empty type. Note, that here the type
   of the only component is not associated with a name.
   Graphically:

    Given: y
    Result:  x  where
                      x =   listTy
                              |
                              y                                            *)
```

```

PROCEDURE MakeTupleType (VAR x : EType;
                         y : ARRAY OF NamedType; len : CARDINAL;
                         fw : FwType);
 (*
   Take the named types in the array 'y' ('len' elements) and create a
   tuple type out of these types. If it has been defined as forward type
   'fw.forward'=TRUE and 'fw.type' contains the empty type. Each component
   type is bound to a name that is stored together with the type.
   Graphically:

    Given:   y[0], ..., y[len-1]
    Result:  x where

              x = tupleTy
                    |
                    |-----------------------
                    |                      |
                   y[0]       ...      y[len-1]
                                                                           *)

PROCEDURE MakeUnionType (VAR x : EType;
                         y : ARRAY OF NamedType; len : CARDINAL;
                         fw : FwType);
 (*
   Take the named types in the array 'y' ('len' elements) and create a
   union type out of these types. If it has been defined as forward type
   'fw.forward'=TRUE and 'fw.type' contains the empty type. Each component
   type is bound to a name that is stored together with the type.
   Graphically:

    Given:   y[0], ..., y[len-1]
    Result:  x where

              x = unionTy
                    |
                    |-----------------------
                    |                      |
                   y[0]       ...      y[len-1]
                                                                           *)

PROCEDURE MakeFctType (VAR x : EType;
                       resultType : EType; resultName : CARDINAL;
                       argTypes : ARRAY OF NamedType; len : CARDINAL;
                       fw : FwType);
 (*
   Take the type 'resultType' and the name 'resultName' and the named types
   in the array 'argTypes' ('len' elements) and create a
   function type out of these types. If it has been defined as forward type
   'fw.forward'=TRUE and 'fw.type' contains the empty type. The 'resultType'
```

```
EveType                                                             Page:   4

   and ‘resultName’ are stored as the first component in the type.
   After this the types of the parameters together with their names are
   stored.
   Graphically:

    Given:   resultType, resultName, argTypes[0], ..., argTypes[len-1]
    Result:  x where

              x = fctTy
                    |
                    |----------------------------------------------
                    |                |                            |
                resultName      argTypes[0]       ...      argTypes[len-1]
                resultType
                                                                           *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*          Test whether a type is a certain compound Eve type             *)
(*                                                                         *)
(* ----------------------------------------------------------------------- *)

PROCEDURE isListType (eType : EType) : BOOLEAN;
 (*
   Test whether the type ‘eType’ is a list type. For that only the top
   level constructor has to be checked (see graphics above).              *)

PROCEDURE isListOf (eTypeList, eType : EType) : BOOLEAN;
 (*
   Test whether the type ‘eTypeList’ is a list type and whether the type
   of the components of this list is ‘eType’.                              *)

PROCEDURE isTupleType (eType : EType) : BOOLEAN;
 (*
   Test whether the type ‘eType’ is a tuple type. For that only the top
   level constructor has to be checked (see graphics above).              *)

PROCEDURE isUnionType (eType : EType) : BOOLEAN;
 (*
   Test whether the type ‘eType’ is a union type. For that only the top
   level constructor has to be checked (see graphics above).              *)

PROCEDURE isFctType (eType : EType) : BOOLEAN;
 (*
   Test whether the type ‘eType’ is a function type. For that only the top
   level constructor has to be checked (see graphics above).              *)

(* ----------------------------------------------------------------------- *)
(*                                                                         *)
(*   Extract information about the components of the compound Eve types   *)
```

```
EveType                                                        Page:   5

(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE GetListType (list : EType;
                       VAR listType : EType);
 (*
   This procedure checks whether `list' is a list type (if not then error)
   and gets the type of the components of the list in `listType'.        *)

PROCEDURE GetFieldType (tuple : EType; name : CARDINAL;
                        VAR nr : CARDINAL; VAR fieldType : EType);
 (*
   This procedure checks whether `tuple' is a tuple type (if not then error)
   and gets the type of the component that is bound to the name `name'
   in `fieldType'. In `nr' the number of this field in the tuple is
   returned (counting starts with 1).                                     *)

PROCEDURE GetTupleTypeList (tuple : EType;
                            VAR typeList : ARRAY OF EType;
                            VAR len : CARDINAL);
 (*
   This procedure checks whether `tuple' is a tuple type (if not then error)
   and gets a list of all component types in `typeList'. In `len' the
   length of this list is returned.                                       *)

PROCEDURE GetVariantType (union : EType; name : CARDINAL;
                          VAR variantType : EType; VAR nr : CARDINAL);
 (*
   This procedure checks whether `union' is a union type (if not then error)
   and gets the type of the component that is bound to the name `name'
   in `variantType'. In `nr' the number of this variant in the union is
   returned (counting starts with 1).                                     *)

PROCEDURE GetVariantTypeByNumber (union : EType; nr : CARDINAL;
                                  VAR variantType : EType);
 (*
   This procedure checks whether `union' is a union type (if not then error)
   and gets the type of the `nr'-th component of the union in `variantType'.
   If `union' has fewer than `nr' components an error occurs.            *)

PROCEDURE GetParTypeList (x : EType;
                          VAR idList : ARRAY OF CARDINAL;
                          VAR typeList : ARRAY OF EType;
                          VAR len : CARDINAL);
 (*
   This procedure checks whether `x' is a function type (if not then error)
   and gets a list of all component types in `typeList' and a list of
   the parameter names in `idList'. In `len' the length of these lists
   is returned. Note that type and name of the return value are not
   delivered by this procedure. To get this information the procedures
```

```
  ‘GetReturnType’ and ‘GetReturnName’ have to be used.                *)

PROCEDURE GetReturnType (x : EType; VAR returnType : EType);
 (*
  This procedure checks whether ‘x’ is a function type (if not then error)
  and gets the type of the return value in ‘returnType’.              *)

PROCEDURE GetReturnName (x : EType; VAR returnName : CARDINAL);
 (*
  This procedure checks whether ‘x’ is a function type (if not then error)
  and gets the name of the return value in ‘returnName’.              *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*                  Test the equality of types                         *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE equalType (x, y : EType): BOOLEAN;
 (*
  Test whether the types ‘x’ and ‘y’ are equal (due to structure
  equivalence).                                                        *)

PROCEDURE TestTypes (typeList1 : ARRAY OF EType; len1 : CARDINAL;
                     typeList2 : ARRAY OF EType; len2 : CARDINAL;
                     VAR ok : BOOLEAN);
 (*
  Test whether ‘len1’=‘len2’ and the types ‘typeList1[i]’ and ‘typeList2[i]’
  are equal for all 1<=i<=len1. Iff this is the case ‘ok’ is set to TRUE.
  This procedure is needed to check whether the types of a function
  in a forward definition and in its actual definition are equal.      *)

PROCEDURE TestTypes1 (x,returnType : EType; typeList : ARRAY OF EType;
                     len : CARDINAL; VAR ok : BOOLEAN);
 (*
  Test whether the type ‘returnType’ is equal to the type of the
  return value of the function type ‘x’ and test whether the type
  ‘typeList[i]’ is equal to the type of the i-th parameter of the
  function type ‘x’. Iff all this is TRUE and the number of parameters
  of the function type ‘x’ is ‘len’ then ‘ok’ returns TRUE.            *)

(* ------------------------------------------------------------------- *)
(*                                                                     *)
(*                Miscellaneous operations on types                    *)
(*                                                                     *)
(* ------------------------------------------------------------------- *)

PROCEDURE SetParNames (name : CARDINAL; idList : ARRAY OF CARDINAL;
                       len : CARDINAL; eType : EType);
 (*
```

```
EveType                                                           Page:   7

    This procedure checks whether `eType' is a function type
    (if not then error). It sets the name of the return value that is
    stored in the type to `name' (overwriting an old value) and it sets
    the names of the parameters that are stored in the type to
    `idList[0]', ..., `idList[len-1]'. The number of the parameters of
    the function type `eType' must be `len' otherwise an error occurs.      *)

PROCEDURE MaxType (x, y : EType; VAR z : EType);
 (*
    This procedure determines the maximal type of the types `x' and `y'
    and returns this type in `z'. The ordering of the types upon this
    procedure is based is the following:

                  x < y  iff  x=anyType AND y<>anyType

    All other types are not comparable (so `anyType' is the bottom element in
    the domain of Eve types). Furthermore, if neither of the two types
    `x' and `y' is `anyType' and `x' and `y' are not equal then a semantic
    error occurs. This procedure is needed in e.g. an IF-THEN-ELSE
    expression where the type of one branch is `anyType' but that of the
    other branch is a specific type `x'. In this case the type of the
    whole expression must be `x' and not `anyType'.                         *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*     Procedures handling the creation of a type file containing a       *)
(*          description of the type of the constant MAIN                  *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE InitTypeFile;
 (*
    This procedure initializes a local data structure that is needed for
    the creation of the type description file, opens the file (``.adt''
    extension!) and writes the first lines into the file that are
    independent of the actual type.                                         *)

PROCEDURE WriteTypeToFile (eType : EType;
                           start : CARDINAL);
 (*
    This procedure transforms the type `eType' from its internal
    representation into a form that is then written to the type description
    file (``.adt'' extension). This type description file consists of numbered
    lines where each line contains a subtype. The letters `I', `C' and
    `B' are standing for the atomar types of integers, characters and
    booleans, respectively. The compound data types are denoted by
    `L' for lists (with one succeeding number denoting the type of the
    components of that list), `T' for tuples (with as many succeeding
    numbers as the tuple has components, describing the types of these
    components) and `U' for unions (with as many succeeding
```

EveType Page: 8

```
   numbers as the union has components, describing the types of these
   components).
   As this procedure is calles recursivley it has an parameter ‘start’ that
   contains the line number of the nest line to be written to the file.
   Example:
    The Eve type ‘tuple’ that is defined by

      IntList == LIST OF INT;

      IBUnion == UNION OF
                  int  == INT,
                  bool == BOOL;

      tuple == TUPLE OF
                 list  : IntList,
                 union : IBUnion;

   will have the following representation in the type description file

     #0:  1         ; This line marks the root of the type
     1:   T  2 4    ; The type is a tuple with components described in 2, 4
     2:   L  3      ; First tuple-component: list (component described in 3)
     3:   I         ; Component of list: integer
     4:   U  3 5    ; Second tuple-comp.: union (comp. described in 3, 5)
     5:   B         ; Second uniuon-component: boolean
                                                                      *)

(* ------------------------------------------------------------------ *)
(*                                                                    *)
(*    Procedure for printing the internal representation of an Eve type   *)
(*                                                                    *)
(* ------------------------------------------------------------------ *)

PROCEDURE PrintType (eType : EType; nr : CARDINAL);
 (*
   This procedure prints the type ‘eType’. The tree structure is
   visiualized with indentations and line numbers denoting the nesting
   level of a node in the type tree (this nesting level is submitted
   by ‘nr’).                                                          *)

END EveType.
```

E.3 Special Modules

```
EveCoIO                                                          Page:   1

DEFINITION MODULE EveCoIO;                 (* Input/Output of the Compiler
=======================================================================
 This module contains some procedure for the input of the Eve program file
 and for the output of the the Adam assembler program and expression files.

 ---------------------------------------------------------------------- *)

FROM EveStringLib IMPORT LongString;
FROM FIO          IMPORT File;

VAR
    success                  : BOOLEAN;    (* Was compilation successful? *)
    fileName                 : LongString; (* filename from command line  *)
    inputFile, asmFile,
    aseFile, adtFile, lisFile : File;      (* File descriptors of input   *)
                                           (* and output files.           *)

(* ---------------------------------------------------------------------- *)
(*                                                                        *)
(*        Procedures for handling the input of the compiler              *)
(*                                                                        *)
(* ---------------------------------------------------------------------- *)

PROCEDURE GetCommandLine;
 (*
   This procedure checks for the correct number of arguments in the
   command line and reads the filename of the input file into the global
   variable 'fileName'. Furthermore, the '.lis' file that mirrors all
   output of the compiler to the screen and includes other protocol
   information is opened in this procedure.                               *)

PROCEDURE OpenInFile;
 (*
   This procedure opens the input file. It takes the filename that has been
   read in the command line, adds the directory of the environment
   variable 'ADDIR' in front and the standard extension at the end of the
   name and then opens the file with that name. The file descriptor
   'inputFile' refers after this procedure to the opened input file.      *)

PROCEDURE CloseInFile;
 (*
   This procedure closes the input file again after the Eve program has been
   entirely read.                                                         *)

PROCEDURE ReadLine (VAR line : ARRAY OF CHAR; VAR len : CARDINAL;
                    VAR eof  : BOOLEAN);
 (*
```

```
EveCoIO                                                                    Page:   2

    This procedure reads one line from the file 'inputFile' and puts that
    line into 'line' (with 'len' as its length). Iff the end of the file
    is reached 'eof' returns TRUE. This procedure is needed in
    'EveGetChar.GetChar'.                                                  *)

(* -------------------------------------------------------------------- *)
(*                                                                      *)
(*        Procedures for handling the output of the compiler            *)
(*                                                                      *)
(* -------------------------------------------------------------------- *)

PROCEDURE OpenAsmFile;
 (*
    This procedure opens the Adam assembler program file ('.asm' extension).
    The name of this file is contained in 'asmName'. It must be in the
    same directory as the input file. The correct extension is automatically
    added to the filename. If such an '.asm' file already exists it is
    renamed to a file with '.bak' instead of '.asm' extension.
    The file descriptor 'asmFile' refers to the '.asm' file after a call
    to this procedure.                                                     *)

PROCEDURE OpenAseFile;
 (*
    This procedure opens the Adam assembler expression file
    ('.ase' extension). This file will contain the code for evaluating the
    result expression of the whole program (defined by the body of MAIN).
    The name of this file is contained in 'aseName'. It must be in the
    same directory as the input file. The correct extension is automatically
    added to the filename.
    The file descriptor 'aseFile' refers to the '.ase' file after a call
    to this procedure.                                                     *)

PROCEDURE OpenAdtFile;
 (*
    This procedure opens the type description file ('.adt' extension).
    This file will contain the type description of the result expression of
    the whole program (defined by the body of MAIN).
    The name of this file is contained in 'adtName'. It must be in the
    same directory as the input file. The correct extension is automatically
    added to the filename.
    The file descriptor 'adtFile' refers to the '.adt' file after a call
    to this procedure.                                                     *)

PROCEDURE OpenLisFile;
 (*
    This procedure opens the protocol file ('.lis' extension).
    This file contains all the information that is also printed to the
    screen during the compilation (i.e. it contains possible errors
    that occur in the compilation) and some other information.
    The name of this file is contained in 'lisName'. It must be in the
```

```
EveCoIO                                                     Page:   3

    same directory as the input file. The correct extension is automatically
    added to the filename.
    The file descriptor 'lisFile' refers to the '.lis' file after a call
    to this procedure.                                                     *)

PROCEDURE CloseAdtFile;
 (*
    This procedure closes the type description file after it has been
    created with a call to 'EveType.WriteTypeToFile'.                      *)

PROCEDURE CloseLisFile;
 (*
    This procedure closes the protocol file at the very end of the
    compilation.                                                           *)

END EveCoIO.
```

```
EveError                                                               Page:   1

DEFINITION MODULE EveError;       (*  Error Handler (used by all 3 passes)
=========================================================================
 This module contains the procedures for the error handling. Especially,
 the error statements are stored locally in this module and printed
 here (only exception: syntax errors; they are printed in ‘EveSyntAn’).
 ------------------------------------------------------------------------ *)

VAR
    synErrors, semErrors  : CARDINAL;  (* Number of syntax/semantic errors *)
    restrictions, semOk   : BOOLEAN;   (* Check if semantics still ok      *)

PROCEDURE LexError    (col, nr : CARDINAL);
 (*
   Print a lexical error, encoded by ‘nr’ in the column ‘col’.
   This procedure is only called in ‘EveLexAn’.                           *)

PROCEDURE SemError    (col, nr : CARDINAL);
 (*
   Print a semantic error, encoded by ‘nr’ in the column ‘col’.           *)

PROCEDURE Restriction (col, nr : CARDINAL);
 (*
   Print a compiler restriction, encoded by ‘nr’ in the column ‘col’.     *)

PROCEDURE CompError   (col, nr : CARDINAL);
 (*
   Print a compiler error, encoded by ‘nr’ in the column ‘col’.           *)

END EveError.
```

```
EveGetChar                                                     Page:   1

DEFINITION MODULE EveGetChar;                          (* Elementary Input
========================================================================
 This module contains procedures for reading the next character from the
 input file (block buffered input) and an altered `WriteLn' procedure that
 counts the line numbers and starts a new page after every 50 lines.
 ------------------------------------------------------------------------ *)

CONST
      endFile =  3C;                                 (*  end-of-text; etx  *)
      endLine = 12C;                                 (*  end-of-line; lf   *)

                                 (* The following Variables are READ-ONLY !! *)
VAR
    nextChar     : CHAR;                (* Character read by GetChar        *)
    charCol      : CARDINAL;            (* column of the character nextChar *)

PROCEDURE GetChar;
 (*
   Reads next Character from source text and stores it in nextChar.
   col is the column number of the character in the printed output (starting
   with 0).
   After each 80-column line, GetChar supplies a special end-of-line
   character endLine.
   Each attempt to read beyond the end of the source text results in
   nextChar=endFile.
   In addition to these elementary tasks, GetChar prints the source text
   with leading line numbers and a page header after every 60th line.
   After a call to GetChar, the line containing nextChar has already been
   printed.                                                                *)

PROCEDURE WriteLn;
 (*
   Starts a new line in the listing. If necessary a new page is started and
   the headline is printed.                                                *)

END EveGetChar.
```

EveWrite Page: 1

```
DEFINITION MODULE EveWrite;            (*  For printing enumerations etc.
==========================================================================
 This module contains procedures for printing the contents of an
 enumeration or a set. Furthermore, it contains a routine that is used
 inside this printing routine but that is also of usage in another module
 (that is why it is exported).
 ------------------------------------------------------------------------- *)

FROM EveStringLib IMPORT LongString;

CONST maxSetLen = 256;

TYPE SetType = SET OF [0..maxSetLen-1];               (* General set type *)

PROCEDURE WriteEnum (enum : CARDINAL; names : ARRAY OF CHAR);
 (*
   This procedure prints the ‘enum’-th substring of ‘names’ where the
   substrings in ‘names’ must be separate by exactly one ’,’. When this
   procedure is called with the ORD of an enumeration value as first
   parameter and a string that contains the names of the components of
   that enumeration, separated by a ’,’, then this procedure prints
   an enumeration.
   This is used mainly for testing.                                      *)

PROCEDURE WriteSet (set : SetType; names : ARRAY OF CHAR);
 (*
   This procedure does the same as the previous procedure but on sets.
   For that a set first has to be transformed into the general set type
   ‘SetType’. After this transformation this procedure can be called with
   a string containing the names of the components in the set.           *)

PROCEDURE GetSubString (enum : CARDINAL; names : ARRAY OF CHAR) : LongString;
 (*
   This procedure gets the ‘enum’-th substring of the string ‘names’,
   where the various substrungs are separated by exactly one ’,’.
   This procedure is mainly used by the two previous procedures but it is
   also used in other modules.                                           *)

END EveWrite.
```

Bibliography

[Agha, 1986]
Gul Agha.
An Overview of Actor Languages.
SIGPLAN Notices, 21(10):58–67, October 1986.

[Aho *et al.*, 1986]
Alfred A. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers — Principles, Techniques and Tools.
Addison-Wesley, 1986.

[Appel and MacQueen, 1987]
Andrew W. Appel and David B. MacQueen.
A Standard ML Compiler.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Appel, 1989]
Andrew W. Appel.
A Runtime System.
Technical report, Princeton University, February 1989.

[Arvind and Nikhil, 1988]
Arvind and R. S. Nikhil.
Executing a Program on the MIT Tagged-Token Dataflow Architecture.
Computation Structures Group Memo 271, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1988.
Also in: PARLE 87, Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, June 15–19, 1987, Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[Arvind *et al.*, 1987]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.
Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.

[Augustsson and Johnsson, 1989]
Lennart Augustsson and Thomas Johnsson.

Parallel Graph Reduction with the $\langle v, G\rangle$-machine.
In *FPCA '89, The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, Imperial College, London, September 11–13, 1989. ACM Press, New York.

[Augustsson, 1987]
Lennart Augustsson.
Compiling Lazy Functional Languages, Part II.
PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Backus *et al.*, 1990]
John Backus, John H. Williams, and Edward L. Wimmers.
An Introduction to the Programming Language FL.
In David A. Turner, editor, *Research Topics in Functional Programming*, pages 219 – 247. Addison-Wesley, 1990.

[Backus, 1978]
John Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM, 21(8):613–641, August 1978.

[Barendregt, 1984]
H. P. Barendregt.
The Lambda Calculus: Its Syntax and Semantics.
North-Holland, 1984.

[Bevan, 1987]
D. I. Bevan.
Distributed Garbage Collection Using Reference Counting.
In *Proceedings of PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[Bird and Wadler, 1988]
Richard Bird and Philip Wadler.
Introduction to Functional Programming.
Series in Computer Science. Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Blaschek *et al.*, 1986]
G. Blaschek, G. Pomberger, and F. Ritzinger.
Einführung in die Programmierung mit Modula-2 (Introduction to Programming in Modula-2)(in German).
Springer, 1986.

[Buchberger, 1985]
Bruno Buchberger.
Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.
In N. K. Bose and D. Reidel, editors, *Recent trends in Multidimensional Systems*, chapter 6, pages 184–232. D. Reidel Publishing Company, Dordrecht-Boston-Lancaster, 1985.

[Burge, 1975]
W. H. Burge.
Recursive Programming Techniques.
Addison-Wesley, Reading, Mass., 1975.

[Burn, 1988]
G. L. Burn.
A Shared Memory Parallel G-machine Based on the Evaluation Transformer Model of Computation.
Draft, GEC Hirst Research Centre, Wembley, UK, November 1988.

[Burn, 1991]
Geoffrey L. Burn.
Implementing the Evaluation Transformer Model of Reduction on Parallel Machines.
Technical Report DOC 90/24, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.
To be published in: *Journal of Functional Programming*, 1(2), April 1991.

[Burstall *et al.*, 1980]
R. M. Burstall, D. B. MacQueen, and D. T. Sannella.
Hope: An Experimental Applicative Language.
In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford University, Stanford, California, August 25–27, 1980. ACM Press.
Also: Internal Report CSR-62-80, Department of Computer Science, University of Edinburgh, May 1980, Update February 1981.

[Carriero and Gelernter, 1990]
Nicholas Carriero and David Gelernter.
How to Write Parallel Programs.
MIT Press, 1990.

[Chandy and Misra, 1988]
Mani K. Chandy and Jayadev Misra.
Parallel Program Design — A Foundation.
Addison-Wesley, Reading, Massachusetts, 1988.

[Clarke *et al.*, 1980]
T. Clarke, P. Gladstone, C. MacLean, and A. Norman.
SKIM — The S, K, I Reduction Machine.
In *Conference Record of the 1980 LISP Conference*, pages 128–135, Stanford University, Stanford, California, August 25–27, 1980. ACM Press.

[Collins, 1975]
George E. Collins.
Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition.
In *Proceedings of the 2nd GI Conference*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, Kaiserslautern, 1975. Springer, Berlin.

[Cooper and Morrisett, 1990]
Eric C. Cooper and J. Gregory Morrisett.
Adding Threads to Standard ML.
Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[Cripps *et al.*, 1987]
M. D. Cripps, John Darlington, A. J. Field, P. G. Harrison, and M. J. Reeve.
The Design and Implementation of ALICE: A Parallel Graph Reduction Machine.
In Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*, pages 300–321. Springer, Berlin, 1987.

[Fairbairn and Wray, 1987]
John Fairbairn and Stuart Wray.
Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Fairbairn, 1985]
J. Fairbairn.
Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus.
Technical report, University of Cambridge Computer Laboratory, Cambridge, UK, May 1985.

[Field and Harrison, 1988]
Anthony J. Field and Peter G. Harrison.
Functional Programming.
International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

[Gehani and Roome, 1986]
N. H. Gehani and W. D. Roome.
Concurrent C.
Software — Practice and Experience, 16(9):842–844, September 1986.

[George, 1989]
Lal George.
An Abstract Machine for Parallel Graph Reduction.
In *FPCA '89, The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 214–229, Imperial College, London, September 11–13, 1989. ACM Press, New York.

[Goldberg, 1988]
Benjamin F. Goldberg.
Multiprocessor Execution of Functional Programs.
Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[Halstead, Jr., 1985]
Robert H. Halstead, Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, October 1985.

[Harper, 1990]
Robert Harper.
Introduction to Standard ML.
Technical report, School of Computer Science, Carnegie Mellon University, September 1990.

[Henderson, 1980]
Peter Henderson.
Functional Programming.
International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[Hoare, 1985]
C. A. R. Hoare.
Communicating Sequential Processes.
Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[Hudak and Fasel, 1992]
Paul Hudak and Joseph H. Fasel.
A Gentle Introduction to Haskell.
Technical report, Department of Computer Science, Yale University, 1992.

[Hudak and Goldberg, 1985]
Paul Hudak and Benjamin Goldberg.
Serial Combinators: Optimal Grains of Parallelism.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 382–399, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Hudak *et al.*, 1991]
Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson.
Report on the Functional Programming Language Haskell.
Research Report 1.1, Department of Computing Science, University of Glasgow, Glasgow, UK, June 1991.
Final Draft.

[Hudak, 1984]
Paul Hudak.
ALFL Reference Manual and Programmer's Guide.
Technical Report YALEU/DCS/TR–322, Second edition, Yale University Department of Computer Science, New Haven, CT, October 1984.

[Hudak, 1986a]
Paul Hudak.
Arrays, Non-Determinism, Side-Effects, and Parallelism: A Functional Perspective (Extended Abstract).
Technical report, Department of Computer Science, Yale University, New Haven, Connecticut, November 1986.

[Hudak, 1986b]
Paul Hudak.
Para-Functional Programming.
IEEE Computer, 19(8):60–70, August 1986.

[Hudak, 1989]
Paul Hudak.
Para-Functional Programming in Haskell.
Technical report, Department of Computer Science, Yale University, June 1989.

[Hughes, 1982]
R. J. M. Hughes.
A Semi-Incremental Garbage Collection Algorithm.
Software — Practice and Experience, 12:1081–1084, 1982.

[Hughes, 1984]
R. J. M. Hughes.
The design and implementation of programming languages.
PhD thesis, Programming Research Group, University of Oxford, 1984.

[Iannucci, 1988]
Robert Alan Iannucci.
A Dataflow / von Neumann Hybrid Architecture.
Technical Report MIT/LCS/TR-418, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[Inmos, 1988]
Inmos.
Occam 2 Reference Manual.
Prentice Hall, New York, 1988.

[Inmos, 1989]
Inmos Limited, Bristol, UK.
3L Parallel C, IMS D711, User Guide, February 1989.

[Jensen, 1988]
Jensen & Partners International.
TopSpeed Modula-2 for IBM Personal Computer and Compatibles, User's Manual, 1988.

[Johnsson, 1987]
Thomas Johnsson.
Compiling Lazy Functional Languages, Part I.
PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Kelly, 1989]
Paul Kelly.
Functional Programming for Loosely-coupled Multiprocessors.
Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[King, 1988]
K. N. King.
TopSpeed Modula-2 for IBM Personal Computer and Compatibles, Language Tutorial.
Jensen & Partners International, 1988.

[Knuth, 1968]
D. E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory, 2(2):127–147, 1968.

[Kranz *et al.*, 1986]
David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams.
ORBIT: An Optimizing Compiler for Scheme.

In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, number 21(7) in SIGPLAN Notices, pages 219–233, July 1986.

[Kranz, 1988]
David Andrew Kranz.
ORBIT: An Optimizing Compiler for Scheme.
Technical report, Yale University, New Haven, Connecticut, 1988.

[Landin, 1964]
P. J. Landin.
The mechanical evaluation of expressions.
The Computer Journal, 6:308–320, 1964.

[Loidl, 1991]
Hans Wolfgang Loidl.
Circular Programs on Compound Data Structures.
Technical Report 92-06, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Loogen, 1990]
Rita Loogen.
Parallele Implementierung funktionaler Programmiersprachen (Parallel Implementation of Functional Programming Languages (in German).
Number 232 in Informatik Fachberichte. Springer, 1990.

[Milner, 1980]
Robin Milner.
A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science.*
Springer, 1980.

[Milner, 1984]
Robin Milner.
A proposal for Standard ML.
In *1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197, Austin, Texas, August 6–8, 1984.

[Nikhil, 1988]
Rishiyur S. Nikhil.
ID (Version 88.0) Reference Manual.
Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

[Papadopoulos and Culler, 1990]
Gregory M. Papadopoulos and David E. Culler.
Monsoon: An Explicit Token-Store Architecture.
In *The 17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.

[Papadopoulos, 1988]
Gregory Michael Papadopoulos.
Implementation of a General Purpose Dataflow Multiprocessor.
PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[Peyton Jones and Lester, 1991]
Simon L. Peyton Jones and David Lester.
A Modular Fully-lazy Lambda Lifter in HASKELL.
Software — Practice and Experience, 21(5):479–506, May 1991.

[Peyton Jones and Lester, 1992]
Simon Peyton Jones and David Lester.
Implementing Functional Languages.
Prentice Hall, 1992.
To appear.

[Peyton Jones and Wadler, 1991]
Simon L. Peyton Jones and Philip Wadler.
A static semantics for HASKELL.
Technical report, Department of Computing Science, University of Glasgow, May 1991.

[Peyton Jones *et al.*, 1987]
Simon L. Peyton Jones, Chris Clack, John Salkild, and Mark Hardie.
GRIP — a High-Performance Architecture for Parallel Graph Reduction.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 98–112, Portland, Oregon, September 14–16, 1987. Springer, Berlin.

[Peyton Jones, 1987]
Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall, New York, 1987.

[Peyton Jones, 1989]
Simon L. Peyton Jones.
Parallel Implementations of Functional Programming Languages.
The Computer Journal, 32(2):175–186, April 1989.

[Peyton Jones, 1991]
Simon L. Peyton Jones.
The Spineless Tagless G-machine: second attempt.
Technical report, Department of Computing Science, University of Glasgow, June 1991.

[Poplet, 1987]
John Poplet.
The Development of a Parallel C Compiler.
Technical report, Definicon Systems, Inc., Edelsbergstr. 8, 8000 München 21, West Germany, 1987.

[Reppy, 1991]
John H. Reppy.
Concurrent Programming with Events.
Reference Manual Version 0.9.6, Department of Computer Science, Cornell University, Ithaca, NY, October 1991.
The Concurrent ML Manual.

[Ruggherio and Sargeant, 1987]
C. A. Ruggherio and J. Sargeant.

Control of Parallelism in the Manchaster Dataflow Machine.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 1–15, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Sarkar and Hennessy, 1986]
Vivek Sarkar and John Hennessy.
Partitioning Parallel Programs for Macro-Dataflow.
In *1986 ACM Symposium on Lisp and Functional Programming*, pages 202–211, Cambridge, Massachusetts, August 4–6, 1986.

[Schmidt, 1986]
David A. Schmidt.
Denotational Semantics — A Methodology for Language Development.
Allyn and Bacon, Boston, 1986.

[Schreiner, 1990]
Wolfgang Schreiner.
ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language.
Master's thesis, Johannes Kepler University, Linz, Austria, October 1990.
Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Schreiner, 1991]
Wolfgang Schreiner.
A Model for the Compilation of Dataflow Languages on Conventional Parallel Architectures.
Technical Report 91-50, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Siegl, 1990]
Kurt Siegl.
Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages.
Master's thesis, Johannes Kepler University, Linz, Austria, 1990.
Also: Technical Report 90-54, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Stoye, 1985]
W. R. Stoye.
The implementation of functional languages using custom hardware.
PhD thesis, Computer Laboratory, University of Cambridge, May 1985.

[Szymanski, 1991]
Boleslaw K. Szymanski, editor.
Parallel Functional Languages and Compilers.
Frontier Series. ACM Press, 1991.

[Takeichi, 1986]
M. Takeichi.
A Functional Machine for Fully Lazy Evaluation (Extended Abstract).
In *Proceedings of the RIKEN Symposium on Functional Programming*, pages 54–60, 1986.

[Takeichi, 1988]
Masato Takeichi.
Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs.
New Generation Computing, 5:377–391, 1988.

[Traub, 1986]
Kenneth R. Traub.
A Compiler for the MIT Tagged-Token Dataflow Architecture.
Technical Report MIT/LCS/TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1986.

[Traub, 1991]
Kenneth R. Traub.
Implementation of Non-Strict Functional Programming Languages.
Research Monographs in Parallel and Distributed Computing. Pitman, London, 1991.

[Treleaven, 1990]
P. C. Treleaven, editor.
Parallel Computers — Object-Oriented, Functional, Logic.
Series in Parallel Computing. John Wiley & Sons, Chichester, 1990.

[Turner, 1975]
D. A. Turner.
An implementation of SASL.
Technical Report 4, University of St. Andrews, 1975.

[Turner, 1976]
David A. Turner.
SASL Reference Manual.
Technical report, University of St. Andrews, December 1976.

[Turner, 1979]
D. A. Turner.
A new implementation technique for applicative languages.
Software — Practice and Experience, 9:31–49, 1979.

[Turner, 1985]
D. A. Turner.
Miranda: A non-strict functional language with polymorphic types.
In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Vollmer, 1990]
Jürgen Vollmer.
Modula-P Reference Manual.
GMD Research Group at the University Karlsruhe, Karlsruhe 1, West Germany, July 1990.

[Wadler, 1985]
P. Wadler.
Introduction to Orwell.
Technical report, Programming Research Group, University of Oxford, 1985.

[Watson *et al.*, 1988]
Ian Watson, John Sargeant, Paul Watson, and Viv Woods.
The Flagship Parallel Machine.
In K. D. Reinartz and Chris Jesshope, editors, *CONPAR 88, Conference on Algorithms and Hardware for Parallel Processing*, Manchester, England, September 12–16, 1988. Cambridge University Press, Cambridge, UK.

[Wentworth, 1989a]
E. P. Wentworth.
Introduction to Functional Programming using RUFL.
Technical Document PPG 89/6, Parallel Processing Group, Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa, May 1989.

[Wentworth, 1989b]
E. P. Wentworth.
The Rhodes University FLFM (Ver. 1.7) — An Abstract Machine for Functional Languages.
Technical Document PPG 89/7, Parallel Processing Group, Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa, May 1989.

[Wikström, 1987]
A. Wikström.
Functional Programming Using Standard ML.
International Series in Computer Science. Prentice-Hall, 1987.

[Wirth, 1984]
N. Wirth.
Compilerbau (Compiler Design) (in German).
Teubner, 1984.

[Wirth, 1988]
Niklaus Wirth.
Programming in Modula-2.
Text and Monographs in Computer Science. Springer, fourth edition, 1988.