Evolving Algebras

J.M.W. Visser

August 6, 1996

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Department of Technical Informatics

Master's Thesis

Afstudeergegevens

Spreker:	J.M.W. Visser
Titel:	Evolving algebras
Datum:	Dinsdag 20 augustus 1996
Tijd:	14:00
Plaats:	Zaal OO, Faculteit Wiskunde en Informatica
Afstudeercie:	Prof. Dr. Ir. F.W. Jansen
	Drs. R. D. Huijsman
	Dr. Ir. R. Sommerhalder
	Ir. J.F.M. Tonino
Samenvatting:	Evolving algebras provide models for arbitrary computational processes, that can at once be viewed as abstract machines, and as formal specifications. Thus, the evolving algebra formalism combines two perspectives on computational processes: specification methods and computational models. Two contributions have been made to the evolving algebra research programme. Firstly, a detailed proposal has been formulated to introduce modules into the evolving algebra formalism. These modules enable structured evolving algebra design, and make new kinds of parallelism available in the theory. Secondly, a tool, called EVADE, has been created for automated support of evolving algebras. This tool comprises both a compiler and a run analyzer. EVADE has been implemented in the functional language Gofer. In the implementation, ample use has been made of a novel programming method, which involves *monads*.

Contents

I The theory of evolving algebras 7

1 Introduction to evolving algebras 10
- 1.1 What are evolving algebras? . . . 10
- 1.2 An example evolving algebra . . . 12
- 1.3 Characteristics and features of evolving algebras . . . 15

2 The core theory of evolving algebras 18
- 2.1 Introduction . . . 18
- 2.2 Runs and the transition graph . . . 20
- 2.3 Identifiers and expressions . . . 21
- 2.4 States . . . 24
- 2.5 The program . . . 26
- 2.6 Transitions . . . 27
- 2.7 Consistency . . . 30

3 Modularization of evolving algebras 35
- 3.1 The purpose of modularization . . . 36
- 3.2 Preparatory adaptations of the theory . . . 36
- 3.3 Modules as parameterized abstractions . . . 37
- 3.4 Evolving algebra modules as parameterized abstractions . . . 42
- 3.5 Functional modules . . . 44
- 3.6 Procedural modules . . . 46
- 3.7 Modules and consistency . . . 51
- 3.8 Modularized evolving algebras in the literature . . . 52
- 3.9 Modules and parallelism . . . 53
- 3.10 Modules and structured evolving algebra design . . . 57

4 A modular evolving algebra for lambda reduction 59
- 4.1 Lambda reduction . . . 59
- 4.2 Graph representation of lambda expressions . . . 61
- 4.3 Modular structure of the evolving algebra . . . 62
- 4.4 Top level module . . . 62
- 4.5 Module for finding the redex . . . 64
- 4.6 Module for reducing the redex . . . 66
- 4.7 Module for constructing the reduct . . . 68

II Automated support for evolving algebras 73

5 Inventory of evolving algebra support **76**
- 5.1 Evolving algebra compilers . . . 76
- 5.2 Evolving algebra run analyzers . . . 77
- 5.3 Interactive theorem provers . . . 77
- 5.4 Evolving algebra transformers . . . 78

6 EVADE: an evolving algebra tool **79**
- 6.1 Overview . . . 79
- 6.2 Example . . . 81
- 6.3 Advanced features . . . 86
- 6.4 Summary . . . 90

7 Comparison of EVADE with other tools **92**
- 7.1 Comparison with respect to the evolving algebras that are accepted. . 93
- 7.2 Comparison of tools within their categories . . . 96
- 7.3 General evaluation . . . 98

8 The programming language Gofer **99**
- 8.1 Functions and function definitions . . . 100
- 8.2 Expressions . . . 101
- 8.3 Types and type definitions . . . 102
- 8.4 Type classes, overloading and polymorphism restriction . . . 104

9 Programming with monads **109**
- 9.1 Monads . . . 110
- 9.2 Monad extensions . . . 116
- 9.3 Monad transformers . . . 121
- 9.4 Monad transformers and extensions . . . 129
- 9.5 Monadic parsers . . . 133
- 9.6 Monadic I/O . . . 138

10 EVADE: monadic implementation in Gofer **144**
- 10.1 Decomposition of EVADE . . . 145
- 10.2 The monads for the various components . . . 148
- 10.3 The exception mechanism of EVADE . . . 150
- 10.4 Implementation of the low level components . . . 151
- 10.5 Implementation of the intermediate level components . . . 161
- 10.6 Implementation of the top level components . . . 166
- 10.7 Extendability of EVADE . . . 169

A Monad Proofs **171**
- A.1 Proofs of the monad laws . . . 171
- A.2 Proofs of the monad extension laws . . . 174
- A.3 Proofs of the monad transformer laws . . . 177

B Evolving algebra specifications **181**

Introduction

Evolving algebras provide models for arbitrary computational processes. They can at once be viewed as abstract machines, and as formal specifications. Thus, the evolving algebra formalism combines two perspectives on computational processes: specification methods and computational models.

Since evolving algebras were first proposed by Yuri Gurevich [Gur91][Gur95], they have been the subject of ongoing research. The graduation project, the results of which are presented in this thesis report, aimed to make a dual contribution to this research. On the one hand, an attempt has been made to advance the development of the *theory* of evolving algebras. On the other hand, a *tool* has been created to support the design and use of particular evolving algebras. These theoretical and more applied efforts have been undertaken conjointly.

The dual focus of the project is reflected in the structure of this thesis report, which consists of two parts. The first part is concerned with the theory of evolving algebras, while the second part is dedicated to the subject of automated support for evolving algebras.

The first part starts with a chapter which presents an informal introduction to evolving algebras. By briefly describing the various elements of evolving algebras, this introduction answers the question what evolving algebras are. As illustrate to answer, an example evolving algebra is presented and explained in detail. An enumeration of the characteristics and distinctive features of evolving algebras concludes the informal introduction.

Chapter 2 presents a more formal exposition of the theory of evolving algebras. All notions constitutive of the theory are explained and defined in turn. This theory is called the *core* theory of evolving algebras, to contrast with the theory of *modular* evolving algebras which is developed in the next chapter.

Chapter 3 opens with a reflection on the purpose of modularization of evolving algebras. The introduction of modules into the theory enables structured design of evolving algebras, and incorporates new kinds of parallelism into the formalism. The development of the modularized theory takes programming language theory as its point of departure, which views modules as parameterized abstractions. This view is applied to evolving algebras and subsequenly elaborated, which leads to the design of two kinds of evolving algebra modules: functional modules and procedural modules. Examples of modularized evolving algebras using modules of these kinds, are presented. Chapter 3 concludes by investigating the benefits of functional and procedural modules with respect to parallelism and structured evolving algebra design.

Chapter 4, the final chapter of part I, contains an elaborate example of a modular evolving algebra. The evolving algebra in question specifies an algorithm for graph reduction of lambda expressions.

The second part of this report is dedicated to automated support for evolving

algebras. In particular, it is concerned with the evolving algebra tool EVADE, which was developed by the author in the course of his graduation project. Two groups of three chapters can be distinguished within part II.

Chapters 5, 6 and 7 assume the standpoint of a user towards EVADE and other tools for automated support. In chapter 5, an inventory is made of conceivable and actually existing evolving algebra tools. These tools are classified into four categories, which are discussed in turn. Chapter 6 contains an exhaustive description of the functionality of EVADE. Both basic and advanced features of the tool are illustrated with examples. As a consequence, this chapter can serve as a tutorial for users of the program. A command summary is supplied for quick reference. In chapter 7, the EVADE is evaluated with respect to other existing tools that are to some extent comparable. The comparison is made both with respect to the evolving algebras accepted by the tools, and with respect to the functionality offered by them.

In the second group of three chapters of part II, the standpoint of the user is abandoned. In these chapters, an exposition is given of EVADE's implementation. The tool is implemented in the functional language Gofer, which is briefly explained in chapter 8. A unique feature of Gofer is its extended type system, which uses constructor classes to support several varieties of polymorphism. In the implementation of EVADE, ample use is made of a novel programming method, which involves *monads*. The monadic programming method is presented and explained in chapter 9. The presentation also contains elaborations of this method, which were developed in the course of implementing EVADE. Apart from monads, the monadic programming method involves monad extensions and monad transformers. Two generic applications of the monadic programming method are presented as well: monadic parsing and monadic I/O. The chapters on Gofer and monadic programming are preliminary to chapter 10, which discusses the implementation of EVADE itself. This last chapter describes a multi-level decomposition of the program. Before the constitutive components of each level are scrutinized, use of monads and the exception mechanism are explained. Finally, the implementation of EVADE is evaluated, and future extensions of the program are anticipated.

Part I

The theory of evolving algebras

The first part of this report is concerned with the theory of evolving algebras. In chapters 1 and 2, a concise informal account and a more elaborate formal treatment are given of the core theory of evolving algebras. In chapter 3, an extension of the core theory is proposed. This extension consists in the introduction of modules into the theory. These modules enable structured design of evolving algebras, and incorporate new kinds of parallelism into the theory. In chapter 4, the modularized theory of evolving algebras is demonstrated by an example. The example is a modular evolving algebra which specifies an algorithm for graph reduction of lambda expressions.

Chapter 1

Introduction to evolving algebras

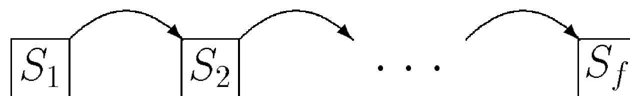
1.1 What are evolving algebras? 10
1.2 An example evolving algebra 12
1.3 Characteristics and features of evolving algebras 15
1.3.1 Characterization of evolving algebras 15
1.3.2 Distinctive features of evolving algebras 16

Evolving algebras provide models of arbitrary computational processes, and can be viewed both as abstract machines and as specifications. Hence, the evolving algebra formalism combines two perspectives on computational processes: specification methods and computational models.

This chapter presents an informal introduction to evolving algebras. Section 1.1 gives a rough characterization of evolving algebras, which answers the question what evolving algebras are. This answer is illustrated by an elaborated example in section 1.2. Section 1.3 briefly recapitulates the characterization of evolving algebras, and lists some distinctive features of the evolving algebra formalism.

1.1 What are evolving algebras?

What are evolving algebras? As a first approximation of the answer to this question we can give the following characterization: An evolving algebra is a state transition system.



Starting from an initial state S_0, a number of states are entered successively, either until a final state is reached, or indefinitely. This rough characterization of evolving algebras immediately gives rise to two questions:

- What are the states of an evolving algebra?
- How are transitions from one state to another brought about?

We will answer these questions in turn.

What are the states of an evolving algebra?

The individual states of evolving algebras are *interpreted vocabularies*. A vocabulary is a collection of names. To interpret a vocabulary means to assign a value to each of the names it contains. (The value assigned to a name is also called its meaning or its interpretation.) For every state of a single evolving algebra the vocabulary is the same. This allows us to speak of *the* vocabulary of an evolving algebra.

The interpreted vocabularies which form the states of evolving algebras are known in mathematics as (many-sorted) algebras, or structures.

How are transitions from one state to another brought about?

The state transitions in an evolving algebra take place on the basis of a *program*. An evolving algebra program is a list of transition rules of the following form:

Program:
if $\langle condition \rangle$
then $\langle update \rangle$
$\vdots$
$\langle update \rangle$
$\vdots$
if $\langle condition \rangle$
then $\langle update \rangle$
$\vdots$
$\langle update \rangle$

The condition in the **if**-clause of a transition rule is called its *guard*. The updates in the **then**-clause together form its update set.

To get from one state to another, the first thing to be done is to determine which of the transition rules in the program are applicable, i.e. which rules are guarded by a condition that is true in the current state S_i. Secondly, by non-deterministic choice a single rule is elected from among the applicable rules. Finally, the updates in the update set of this elected rule are fired *in parallel* at the current state to produce the new state S_{i+1}.

When this procedure is carried out on the initial state S_0 and subsequently on each new state, a chain of transitions is created.

From the brief description of evolving algebras just given, one can already glean what ingredients are needed to define an evolving algebra. These ingredients are the following:

- A vocabulary or, more generally, a signature.
- A description of the initial state S_0.
- A program.

The vocabulary lists the names of the evolving algebra. Since each of these names is intended to name values of a particular type only, the description of a vocabulary can also include type information. Further, distinctions can be made between names of sets and names of functions, and between names whose interpretation can change during transitions, and names whose interpretation is invariant. A vocabulary augmented with type information and information about these distinctions is called a signature.

The description of the initial state of an evolving algebra can take several forms. The most straightforward form of initial state description is a mapping of names to values. Alternatively, one can describe an initial state by supplying a mapping to values for only a portion of names. This mapping is then supplemented by a start update set which assigns values to the remaining names by performing updates.

The program of an evolving algebra definition lists the transition rules of the evolving algebra.

The information that is declared explicitly in the vocabulary or signature, is also implicitly present in the program and the initial state description. As a consequence, the vocabulary or signature of the evolving algebra definition can be suppressed. However, we prefer to make all three defining elements explicit.

1.2 An example evolving algebra

As an illustration to the brief description of evolving algebras given in the previous section, we will now consider an example evolving algebra. Our example evolving algebra specifies an algorithm for creating a linked list of length n, which holds the successive factorials $0! \ldots n!$. We will first consider the start update set and the program:

Start:

$i := 0$
new $e : ListElem$ **with**
  $head(e) := 1$
  $root := e$
  $last := e$

Program:

if $i < n$
then $i := i + 1$
    new $e : ListElem$ **with**
      $head(e) := head(last) * (i + 1)$
      $last := e$
      $tail(last) := e$

In the start update set the counter i is initialized to 0, and the first element of the linked list is created. The "pointers" *root* and *last* are initialized to point to this first element. The value 1 is assigned to the head of the new element, thus filling in the first position in the linked list.

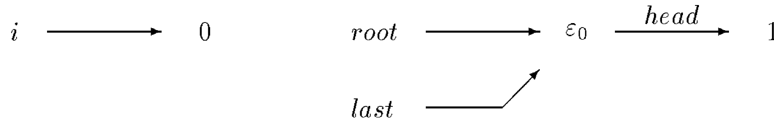


Figure 1.1: Example EA: Initial state.

The program of this example consists of a single transition rule only. The guard of this rule tests whether the counter i has reached its final value n yet. If not, the update set of the transition rule will be fired. The updates in this set increment the counter i, and create a new element of the linked list. The pointers $tail(last)$ and $last$ are updated to point to this new element. Also, a pointer *head* is established from the new element to the new factorial value. Remember that all of these updates are performed in parallel, nested updates as well as the others. If they had been written down in another order, their effect would not have been any different. For instance, the argument for which the tail pointer is locally updated is the old value of *last*, not the new one.

This example demonstrates two forms of updates that can appear in evolving algebras.

- Updates of the form $a := b$ are called *local function updates*.
- Updates of the form **new** $a : D$ **with** *updates* are called *new-updates* or *extension updates*.

Local function updates are similar to assignments in imperative programming languages. Only the former are more general than the latter, because they allow assignment to function names, not only to variables. New-updates perform a kind of allocation. They introduce a new element and contain nested updates in which the new element can be referred to by a new, local name. Apart from these two forms of update that appear in our example, there is one more form: the remove-update. For the moment, this third form of update, which performs a sort of deallocation, will be left out of consideration.

We will now trace the execution of our example evolving algebra. Imagine the value of n to be 3. Firing the start update-set produces the following initial state[1]:

$$\begin{array}{lll} i & \Rrightarrow & 0 \\ root & \Rrightarrow & \varepsilon_0 \\ last & \Rrightarrow & \varepsilon_0 \\ head & \Rrightarrow & \phi[\varepsilon_0 \mapsto 1] \end{array}$$

The triple arrow in $n \Rrightarrow v$ indicates that the name n has the value v. The subscripted epsilon ε_i denotes an anonymous element of the domain *ListElem*. The function ϕ denotes a partial function that is undefined over its entire domain. The notation $f[a \mapsto v]$ refers to the function which is equal to f except in the argument a, for which its value is v. A graphical depiction of this state is given in figure 1.1.

This state is obtained by performing in parallel the updates in the start update set. The first updates initializes i to 0. The second update introduces a new element ε_0, assigns in parallel this element to *root* and *last* and locally (at the argument e_0) initializes the function *head* to 1.

Next, the program of the evolving algebra is executed. Starting from the initial state S_0, we will first enter the state S_1 (see figure 1.2):

[1] Actually, only the part of the state is shown, which can be subject to change.

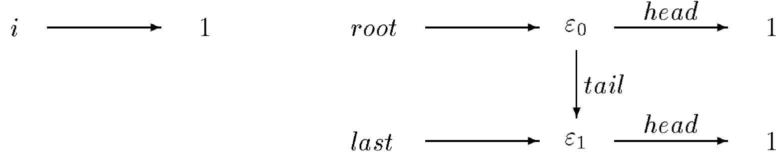


Figure 1.2: Example EA: state after first transition.

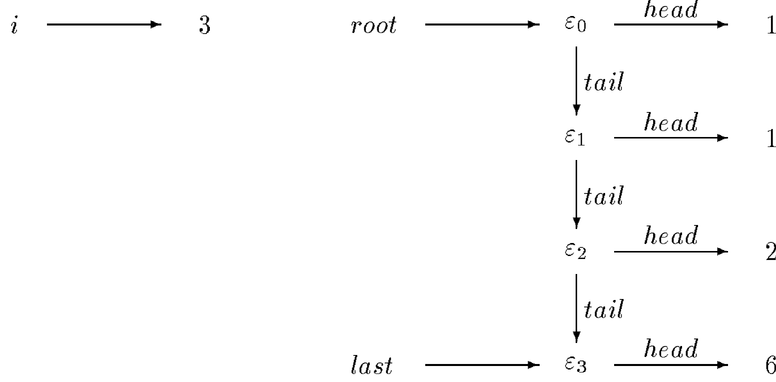


Figure 1.3: Example EA: final state.

$$\begin{array}{lcl} i & \Rrightarrow & 1 \\ root & \Rrightarrow & \varepsilon_0 \\ last & \Rrightarrow & \varepsilon_1 \\ head & \Rrightarrow & \phi[\varepsilon_0 \mapsto 1, \varepsilon_1 \mapsto 1] \\ tail & \Rrightarrow & \phi[\varepsilon_0 \mapsto \varepsilon_1] \end{array}$$

This state is obtained by application of the only transition rule in the program, i.e. by performing this rule's updates in parallel. The first of these updates increments the value of i to 1. The second update introduces a new element. The first of the nested updates of the second update uses the old value of i to calculate the value it assigns locally to *head*. The second nested update redirects the *last* pointer to the new element. The third nested update establishes a *tail* pointer from the former last element to the new element.

Repetitive execution of the program will bring us through a third state S_2 to the fourth and final state of the evolving algebra S_3 (see figure 1.3):

$$\begin{array}{lcl} i & \Rrightarrow & 3 \\ root & \Rrightarrow & \varepsilon_0 \\ last & \Rrightarrow & \varepsilon_3 \\ head & \Rrightarrow & \phi[\varepsilon_0 \mapsto 1, \varepsilon_1 \mapsto 1, \varepsilon_2 \mapsto 2, \varepsilon_3 \mapsto 6] \\ tail & \Rrightarrow & \phi[\varepsilon_0 \mapsto \varepsilon_1, \varepsilon_1 \mapsto \varepsilon_2, \varepsilon_2 \mapsto \varepsilon_3] \end{array}$$

This third state is the final state, because the guard $i < n$ of the only transition rule of the program is false in this state (remember that we chose $n = 3$). Hence there are no applicable rules in this state. Thus, it is a final state.

Above, we have only given the program of our example evolving algebra, and its start update set. Hence, we still need to provide two things to complete the definition of the evolving algebra. Firstly, we need to provide a vocabulary or signature. The information in this signature — what names the vocabulary contains, and what types

these names have — is already implicitly present in the program. Secondly, we need to supplement the start update set with a mapping of names to values to make the description of the initial state complete. The signature and the mapping can be presented in combination as follows:

Signature:

static sorts

$N \Rrightarrow \mathbb{N}$

dynamic sorts

$ListElem$

static functions

$n : N \Rrightarrow 3 : \mathbb{N}$ (bijvoorbeeld)

$+ : N \times N \rightarrow N \Rrightarrow + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$* : N \times N \rightarrow N \Rrightarrow \cdot : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

dynamic functions

$i : N$

$head : ListElem \rightarrow N$

$tail : ListElem \rightarrow ListElem$

$root : ListElem$

$last : ListElem$

The signature groups the names of the vocabulary into four categories:

- Static sort names. In the example, N is a static sort name. It is invariantly interpreted as the mathematical set of natural numbers $\mathbb{N}$.

- Dynamic sort names. $ListElem$ is a dynamic sort name. Initially, $ListElem$ is interpreted as an empty set. The new-updates in the start update set and the program enlarge this set with one element in each transition.

- Static function names. The names n, $+$ and $*$ are static function names. The first one names a 0-ary function (constant), the other two name binary functions. They are invariantly interpreted as the number three, addition of natural numbers and multiplication of natural numbers, respectively.

- Dynamic function names. The names i, $head$, $tail$ and $root$ are dynamic function names. They are initially interpreted as Undef and the function ϕ, which is undefined over its entire domain. Their interpretation changes during the evolution of the evolving algebra under influence of the various updates. Note that dynamic functions are not necessarily of dynamic sort.

The values of static sort and names are given by a mapping of names to values. The values of dynamic sorts and functions are established by the start update set, before which they are assumed to be empty sets and completely undefined partial functions, respectively.

1.3 Characteristics and features of evolving algebras

1.3.1 Characterization of evolving algebras

At the top of section 1.1 we gave a rough characterization of evolving algebras by stating that evolving algebras are state transition systems. The ensuing account of

what the states of evolving algebras are, how state transitions are brought about, and what ingredients are needed to define an evolving algebra, served to add detail to this rough characterization. The example given in section 1.2 illustrated these matters further, and showed what kinds of update are possible on evolving algebra states. We can summarize these matters in four points:

- Evolving algebras are state transition systems.
- The states of evolving algebras are interpreted vocabularies. The names in these vocabularies can be of different types, and fall into four different categories. The values to which the names are interpreted are mathematical sets and functions.
- The control-flow of an evolving algebra consists of (1) a single loop of repetitive execution of the program, and (2) conditions guarding the transition rules in the program. Since several guarding conditions can be true simultaneously, the control flow of an evolving algebra is possibly non-deterministic.
- There are three kinds of updates: local function updates, new-updates and remove-updates. These kinds of updates perform general assignment, allocation and deallocation, respectively. The updates in a (nested) update set are performed in parallel.

These four points constitute an informal description of the evolving algebra formalism. In chapter 2 a more formal and elaborate account will be given.

1.3.2 Distinctive features of evolving algebras

As has been explained in the foregoing sections, the states of evolving algebras are interpreted vocabularies. The values to which the names in these vocabularies are interpreted can be arbitrary mathematical functions and sets. As a consequence, the evolving algebra formalism puts the entire mathematical realm at the disposal of the evolving algebra designer. This is a source of some agreeable features of evolving algebras.

Variable level of abstraction In the evolving algebra formalism, there is no fixed collection of basic building blocks with which the evolving algebras designer must make do. In stead, one can import any mathematical function or set that suits one's needs. As a consequence, the level of abstraction of evolving algebras is variable, and can be adjusted to suit the abstraction level of the computational process to be modelled. No encoding of the process and its data is needed.

Low methodological overhead To start using evolving algebras, one does not need to be initiated to a completely new language. A minimal mathematical background is assumed, to which a small collection of new language constructs is added. These new language constructs are imperative in style, which makes them readily accessible. As a consequence, the evolving algebra formalism is easy to learn, to use, and to understand.

Universality According to Gurevich' evolving algebra thesis, any kind of computational process can be modelled with evolving algebras. In principle, there is no limit on their applicability. They are at least as powerful as Turing machines.

Executability When the import of sets and functions into evolving algebras is restricted to computable ones, evolving algebras can be executed. Hence, they can be used as a programming language as well as a specification language. In fact, the evolving algebra formalism constitutes a programming language that combines the formal rigor of functional languages with the perspicuity of imperative languages.

Thus, evolving algebras offer simplicity and modelling power at any desired level of abstraction.

Chapter 2

The core theory of evolving algebras

2.1 Introduction . . . 18
2.1.1 Methodological remarks . . . 18
2.1.2 Structure of the theory . . . 19
2.1.3 Structure of the chapter . . . 20
2.2 Runs and the transition graph . . . 20
2.3 Identifiers and expressions . . . 21
2.3.1 Identifiers . . . 21
2.3.2 Expressions . . . 22
2.4 States . . . 24
2.5 The program . . . 26
2.6 Transitions . . . 27
2.6.1 Applicability of rules . . . 28
2.6.2 Effects of updates . . . 28
2.7 Consistency . . . 30
2.7.1 Purpose of a notion of consistency . . . 30
2.7.2 Sources of inconsistency . . . 31
2.7.3 Definition of consistency . . . 33

2.1 Introduction

2.1.1 Methodological remarks

In the foregoing chapter, an informal description was given of evolving algebras. In this chapter, we will give a formal exposition of the core theory of evolving algebras. This account is based on two introductory articles written by Yuri Gurevich *Evolving Algebras: A Tutorial Introduction* [Gur91] and *Evolving Algebras 1993: Lipari*

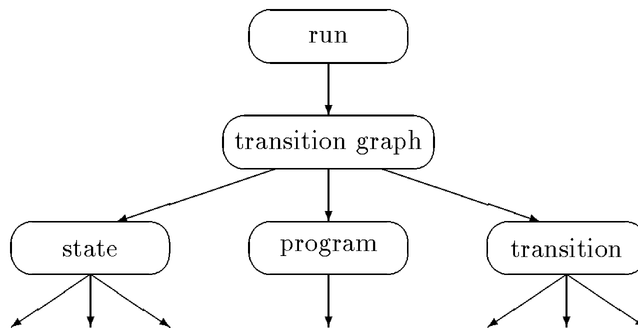


Figure 2.1: Definitional structure of the theory of evolving algebras.

Guide [Gur95]. Also, the section on evolving algebras in the article *A Mathematical Definition of Full Prolog* [BR94, §1] by Egon Börger and Dean Rosenzweig has been consulted. The main source, however, is Hans Tonino's report by the title *A formalization of many-sorted evolving algebras* [Ton93].

The several treatments of evolving algebras in these four sources diverge in a number of respects. In the first place, there is a divergence in terminology. For some concepts different names are employed, and some names are employed for different concepts. The terminology of the present paper has been assembled as much in accordance with the sources as possible, while aiming for maximal clarity and orthogonality.

Secondly, the sources diverge in the amount of symbolism and the degree of formal rigor. Tonino's treatment of evolving algebra is a full formalization, involving a large amount of symbolism and a substantial number of formal definitions and formal theorems. In this introductory chapter, we will try to give a *formal* presentation of evolving algebras in *natural* language; symbolism will not be used *complementary*, but only *supplementary* to natural language. To guide the reader through this formal presentation informal speech will not be eschewed.

Thirdly, Gurevich' articles deal exclusively with one-sorted evolving algebras, while Tonino's report is dedicated to many-sorted evolving algebras. The Börger-Rosenzweig article, finally, is ambivalent between a one-sorted and a many-sorted approach, due to the informal treatment of its notion of *universes*. In this chapter, we will give an account of many-sorted evolving algebras, for this is the more general approach from which the one-sorted case can be derived.

2.1.2 Structure of the theory

The purpose of the theory of evolving algebras is to provide operational semantics for arbitrary algorithms, i.e. to provide models for computational processes. This goal is achieved by modelling a computation as a *run* of an evolving algebra. In other words, the theory is meant to explain the phenomenon of *computation*, and the explanans it offers for this explanandum is the notion of a *run*.

Thus, the notion of a *run* constitutes the focal point of the core theory of evolving algebras. All other notions of the theory are ancillary to it; they derive their right of existence solely from their contribution to the definition of this central notion. In figure 2.1 the notions of the theory have been depicted as nodes in a tree-like structure. The edges in this graph lead from definiendum to definiens. For example, the notion

of a *transition graph* is defined in terms of the notions of a *state*, of a *state transition*, and of a *program*. Thus, figure 2.1 depicts part of the definitional structure of the theory of evolving algebras. The root node, which is no definiens of any other notion, is of course the notion of a *run*.

The definitional structure of the theory is not depicted in its entirety in figure 2.1. Only the upper part of the structure is shown. The bottom part has been pruned away, because too many nodes and arrows would clutter up the picture. In later sections, some portions of the hidden part will be depicted separately. Among the omitted nodes are the direct definientes of the notions of *state*, *program*, and *transition*. Also, the notions of various groups of *identifiers* and *expressions*, which form the base of the definitional structure, are hidden.

The notion of an *evolving algebra* is absent from the definitional structure, which may come as a surprise to the reader. In the informal account of the preceding chapter, we actually used the term "evolving algebra" ambiguously between, on the one hand, a *run*, and on the other hand, the triple of elements needed to specify a run: the signature, the initial state description and the program. In this chapter, we will preserve the latter meaning of the term. Thus, an evolving algebra can be defined as a triple of a *signature*, a *program* and an initial *state*. The three notions involved in this definition, *signature*, *program*, and *state* are each ancillary to the definition of the notion of a *run*, but the notion of an *evolving algebra* itself is not. The utility of this latter notion lies in the *grouping together* of the elements that are necessary for *specifying* a particular run. But for the purpose of *specification*, the three elements will do quite well ungrouped. The notion of an *evolving algebra* is therefore not included in the definitional structure of which the notion *run* is the root.

Strictly speaking then, the initial state, the program and the signature are the only three components of an evolving algebra, while the other states, the state transitions, the transition graph and the runs are not actually part of the evolving algebra, but merely induced by it. Still, speaking somewhat informally, we will simply speak of the states of the evolving algebra in stead of the states induced by the algebra, and likewise for the other entities induced by the evolving algebra.

2.1.3 Structure of the chapter

The organization of the exposition of the theory of evolving algebras in this chapter can be outlined by reference to the definitional structure. In section 2.2 the stem of the definitional tree will be discussed, containing the notions of a *run* and of a *transition graph* (see figure 2.1). Next, section 2.3 will deal with the basis of the definitional structure, which contains notions of various groups of *identifiers* and *expressions*. Then, the three subtrees below *state*, below *program*, and below *transition* will successively be discussed in sections 2.4, 2.5, and 2.6, thus bridging the gap between the stem and the base of the definitional tree. Finally, in section 2.7, a definition will be constructed of a notion of consistency for evolving algebra update sets.

2.2 Runs and the transition graph

We will first concern ourselves with the stem of the definitional structure. As can be seen from figure 2.1, the notion of a *run* is defined in terms of the notion of a *transition graph*. A *transition graph*, in its turn is defined in terms of *states*, *programs*, and *transitions*, as follows:

Definition A *transition graph* is a directed graph the nodes of which are states and the edges of which are transitions effected by the program.

This definition can be clarified as follows. Applied to an initial state, a program can effect transitions to one or more successor states. Application of the program to these successor states can effect transitions to further states, and so on, possibly indefinitely. Thus, repeated application of the program amounts to the spinning of a possibly infinite web of states connected by transitions: a transition graph.

In the upcoming sections, the definientes *state*, *program*, and *transition* of the notion *transition graph*, will themselves be defined. Also, it will be made clear under what conditions a program effects a transition from one state to another.

Given the notion of a *transition graph*, we can define the notion of a *run* as follows:

Definition A *run* is a path through a transition graph which starts at a certain node, and which ends, if it ends, in a node that has no outgoing edges.

A run can be either an infinite path, in which case the modelled computation is non-terminating, or it is a finite path which ends in a dead-end. The first node of a run is called the initial state. Thus, a run is a chain of states connected by transitions which are effected by a program P:

$$S_i = S_1 \xrightarrow{P} S_2 \xrightarrow{P} \cdots \xrightarrow{P} S_n$$

or, in case of non-termination:

$$S_i = S_1 \xrightarrow{P} S_2 \xrightarrow{P} \cdots$$

where S_i is the initial state of the run.

We now turn form the stem of the definitional structure of the theory of evolving algebras to its base.

2.3 Identifiers and expressions

In the description of the states and the program of an evolving algebra a number of *identifiers* is used, with which *expressions* are built. These notions will be explained in turn in the following two subsections.

2.3.1 Identifiers

The identifiers are gathered in a *vocabulary*.

The vocabulary, enhanced with some additional information *about* the identifiers it contains, constitutes a *signature*. This additional information consists in restrictions on what kind of entity each identifier can be used to identify. More specifically, the following additional information is stored in the signature:

- Which identifiers are used to name *sorts*, which are used to name *functions*, and which are used to name *variables*.
- Which sort and function identifiers are *dynamic*, which *static*.
- What *signature* the function named by each function identifier must have.

- What the sort of each variable is.

At this point, we can not yet elaborate upon the static-dynamic distinction of identifiers. In the upcoming sections it will become clear that static identifiers have the same interpretation in every state, while dynamic identifiers can have different interpretations in different states.

Thus, a *signature* can be defined as follows:

Definition A *signature* is a sept-tuple:

$$\langle \text{SSort, DSort, SFun, DFun, FunSigs, Vars, VarSorts} \rangle$$

where

- Ssort, Dsort, SFun, DFun and Vars are pairwise disjoint collections of static sort identifiers, dynamic sort identifiers, static function identifiers, dynamic function identifiers and variables, respectively.
- FunSigs is a function which maps function symbols to their signatures.
- VarSorts is a function which maps variables to their sorts.

When we actually describe a particular signature, we will not present it as a sept-tuple. In stead, we will present it in the form of four lists, which respectively contain static sort identifiers, dynamic sort identifiers, static function identifiers together with their signatures, and dynamic function identifiers together with their signatures. The variables and their sorts will not be mentioned at all, but will remain implicit in the program. Often, we will also integrate into the presentation of the signature a partial description of the initial state. The signature of the example in section 1.2 was presented in this form.

2.3.2 Expressions

Using the identifiers from the vocabulary, we can formulate expressions. Three groups of expressions can be distinguished: terms, updatable terms, and conditions. The second group is a subgroup of the first.

Terms

Variables and zero-ary function identifiers are simple terms. Complex terms are built by providing n-ary functions with n arguments of the appropriate sort. Thus, terms are recursively defined as follows:

Definition

1. If v is a variable of sort s, then v is a *term* of sort s.
2. If f is a function symbol of signature $() \rightarrow s$, then f is a *term* of sort s.
3. If f is a function symbol of signature $s_1 \times s_2 \times \cdots \times s_n \rightarrow s$ $(n > 0)$, and $t_1, t_2, \ldots, t_n$ are terms of sort $s_1, s_2, \ldots, s_n$, then $f(t_1, t_2, \ldots, t_n)$ is a *term* of sort s.

Updatable terms

A special subgroup of terms are called *updatable terms*. The variables are included in this group, as well as all the terms that have a *dynamic* outermost function identifier. Updatable terms can be recursively defined as follows:

Definition

1. If v is a variable of sort s, then v is an updatable term of sort s.

2. If f is a dynamic function symbol of signature $() \rightarrow s$, then f is an updatable term of sort s.

3. If f is a dynamic function symbol of signature $s_1 \times s_2 \times \cdots \times s_n \rightarrow s \ (n > 0)$, and $t_1, t_2, \ldots, t_n$ are terms of sort $s_1, s_2, \ldots, s_n$, then $f(t_1, t_2, \ldots, t_n)$ is an updatable term of sort s.

As we will see in section 2.5 on programs, only updatable terms can be used as subjects in updates; non-updatable terms are excluded from performing this role.

Conditions

Using the usual boolean constants *true* and *false*, the usual boolean connectives $\neg$, $\wedge$, $\vee$, the equality sign for terms $=$, and the predicate *defined(t)* on terms, we can build boolean expressions from terms. These boolean expressions are called *conditions*. Thus, conditions can be recursively defined as follows:

Definition

1. *true* and *false* are *conditions*.

2. If t is a term, then *defined(t)* is a *condition*.

3. If t_1 and t_2 are terms , then $t_1 = t_2$ is a *condition*.

4. If c is a condition, then $\neg c$ is a *condition*.

5. If c_1 and c_2 are conditions, then $c_1 \wedge c_2$ and $c_1 \vee c_2$ are *conditions*.

6. If t is a truth-valued term, then t is a *condition*.

The predicate *defined(t)* evaluates to true only if the term t has a denotation other than Undef according to the interpretation given by the current state.

As we will see in section 2.5 on programs, conditions will be used as *guards* in rules. The applicability of a rule will depend on the value of the condition that is its guard.

The identifiers and the expressions form the base of the definitional structure of the theory of evolving algebras. The stem of this structure was described in section 2.2. In the following section we will build the first of three bridges that will close the gap between stem and base.

2.4 States

In the foregoing section, the *syntax* of three kinds of expressions was presented. In this section, their *semantics* will be given. The syntactical description explains how expressions such as terms, updatable terms and conditions are composed from the identifiers in a vocabulary. As the word itself reveals, an expression, once built, can be used to *express* something. What is expressed is called the semantic value, or the *meaning* of the expression. The association of meanings to expressions is called an *interpretation* of those expressions. In this section it will be made clear how the meanings of expressions are determined, i.e. what constitutes an interpretation of them.

As an expression is built from certain identifiers, its meaning is composed from the meanings of the identifiers. Hence, to determine the meanings of all constructible expressions, all that needs to be given are the meanings of the identifiers in the vocabulary. In other words, an interpretation of the identifiers completely fixes the values of the expressions formed with them. If the interpretation of identifiers changes, so do the values of expressions.

An interpretation of the identifiers of the vocabulary is also called a *state*, which brings us to the following definition:

Definition A *state* is an association of meanings to the sort identifiers, function identifiers, and variables in the vocabulary.

Hence, different interpretations of the identifiers in the vocabulary give rise to different states. However, the states of a single evolving algebra are not allowed to differ in the interpretation of static identifiers, but only in the interpretation of dynamic identifiers and variables.

The vocabulary of an evolving algebra contains three different groups of identifiers: sort identifiers, function identifiers and variables. Likewise, its states can each be divided into three interpretations: the interpretation of sort identifiers, the interpretation of function identifiers and the interpretation of variables.

Interpretation of sort identifiers The meaning of a sort identifier is a set. Hence, the sorts of evolving algebras are sets.

Interpretation of function identifiers The meaning of a function identifier is a (partial) function of with signature which accords with the signature specification in the vocabulary.

In the special case of a zero-ary function identifier, the meaning is a zero-argument — and therefore constant — function that returns an element of the codomain. In stead of this zero-argument function, one can take the element that it returns as the meaning of the identifier.

Interpretation of variables The meaning of a variable is an element of the set which is its sort.

Every sort identifier must have a meaning in every state; uninterpreted sort identifiers are not allowed. Uninterpreted function identifiers are likewise not admitted, but function identifiers are allowed to be interpreted as functions that are only partially defined on the domain indicated by the signature. Consequently, some expressions built from these function identifiers will be without meaning. The last category of

identifiers, the variables, do not need to have a meaning in every state; they are allowed to be simply uninterpreted.

Only the meanings of identifiers and not of the expressions constructible from them, are given in a state. However, the meanings of the expressions are determined by the meanings of the identifiers from which they are built, and can be derived from them by the process of evaluation. To reflect the distinction between what is explicitly given in a state and what is merely determined by it through the process of evaluation, we will call the meaning of an identifier its *interpretation*, and the meaning of an expression (term, updatable term, or condition) its *value*. Since the meanings of expressions are thus implicitly given in a state, we can speak of of the meaning of an expression in a state equally as of the meaning of an identifier in a state.

Evaluation occurs in the usual manner. From the interpretation of identifiers, the values of terms are determined, and from the values of terms the values of conditions are determined. In general, an expression is undefined if any of its composing identifiers is. Conditions of the form $defined(t)$ and $t_1 = t_2$ are exceptions to this rule. For clarity, we will explicitly list our rules of evaluation. In order to make our treatment more fluid, we will assume that uninterpreted identifiers and undefined terms have a meaning after all: the special element Undef.

The evaluation of terms in a state S proceeds according to the following recipe:

- The value of a term v in state S is the interpretation of the variable v in S. If the interpretation of the variable is Undef, so is the value of the term.
- The value of a term f in state S is the interpretation of the 0-ary function identifier f in S. If the interpretation of the function identifier is Undef, so is the value of the term.
- The value of a term $f(t_1 \ldots t_n)$ in state S is the interpretation of the n-ary function identifier f in state S, applied to the tuple of values of the terms t_i in S, provided none of the terms t_i have the value Undef in S. Otherwise, the value of the term $f(t_1 \ldots t_n)$ is Undef.

The evaluation of conditions proceeds along the following guidelines:

- The values of $true$ and $false$ in S are true and false, respectively.
- The value of $defined(t)$ in S is false if t has the value Undef in S. Otherwise, the value is true.
- The value of $t_1 = t_2$ in S is true if the values of t_1 and t_2 in S are equivalent according to some definition of equivalence of values of this sort. If t_1 and t_2 do not denote equivalent values or have the value Undef in S, then the value is false.
- The value of $\neg c$ in S is true if the value of c in S is false, and false if it is true.
- The values of $c_1 \wedge c_2$ and $c_1 \vee c_2$ in S are determined by the values of c_1 and c_2 according to the usual truth tables for logical conjunction and disjunction, respectively.
- The value of the condition t in S is the value of the truth-valued term t in S. The value of the term is not allowed to be Undef.

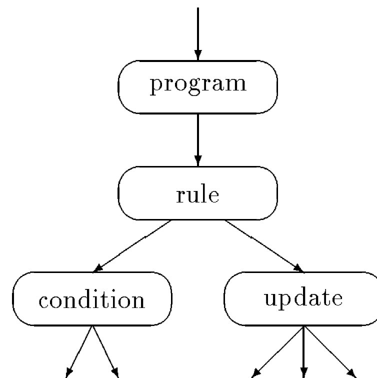


Figure 2.2: Definitional structure below the notion of *program*.

Hence, all functions in an evolving algebra are strict in all their arguments. Terms that contain uninterpreted variables or function identifiers that are not defined for the arguments they receive, are likewise without meaning. There is only one exceptional function, which is not strict in its argument: the special predicate *defined(t)*, which evaluates to false if the term t does not have a meaning in the current state, and otherwise to true.

The notion of *state* connects the base of the definitional structure of the theory of evolving algebras to its stem. In the following two sections, two more connections will be made.

2.5 The program

According to the definitions in section 2.2 the nodes of a transition graph are states. These were defined in the foregoing section. The edges of a transition graph are *transitions*. Because the transitions are *effected* by the program, in this section we will consider the notion of a *program* before we turn to the notion of *transition* in the following section. The portion of the definitional structure below the notion of *program* has been depicted in figure 2.2.

A program consists of a number of transition rules that each specifies a set of updates and a condition for their application. Thus, a program is defined in terms of rules as follows:

Definition A *program* is a finite set of transition rules.

Rules, in turn, are defined in terms of *conditions* and *updates*:

Definition A *rule* consists of a condition and a finite set of updates.

To specify a rule, the following syntax is used:

if ⟨*condition*⟩
then ⟨*updates*⟩

The condition that appears in a rule is also called its *guard*. It indicates when the rule is *applicable*.

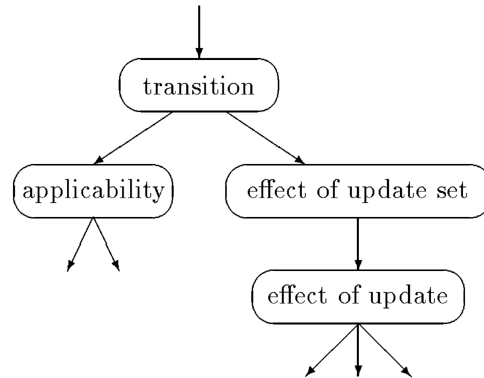


Figure 2.3: Definitional structure below the notion of *transition*.

There are three possible kinds of updates: *local function updates*, *extend updates* and *remove updates*. We will consider each of these kinds of updates in turn:

Local function update To specify a local function update, the following syntax is used:

$$\langle \textit{updatable term} \rangle := \langle \textit{term} \rangle$$

The updatable term indicates the *subject* of the update.

Extend update To specify an extend update, the following syntax is used:

new ⟨*variable*⟩ : ⟨*dynamic sort identifier*⟩ **with**
⟨*updates*⟩

The sort of the variable must be be in accordance with the dynamic sort identifier. The nested updates are again finite in number and can again be of all three kinds.

Remove update To specify a remove update, the following syntax is used:

rem ⟨*updatable term*⟩ : ⟨*dynamic sort identifier*⟩

The sort of the updatable term must again be in accordance with the dynamic sort identifier.

Two connections between the stem and the base of the definitional structure of the theory of evolving algebras are now in place. In the next section, the third and final connection will be made.

2.6 Transitions

In the previous section, programs were described purely as syntactical objects. In the current section, we will provide them with a semantics. The semantics of a program consists in its effect upon states, which are *transitions*. The notion of a transition is the third definiens of *transition graph*, as can be seen from figure 2.1 and the definition

in section 2.2. The portion of the definitional structure below this notion has been depicted in figure 2.3.

A transition is a relation between two states, indexed by the program. In section 2.2 the notation $S_i \xrightarrow{P} S_{i+1}$ has already been used for this relation. We define the relation by giving the conditions under which it obtains.

Definition A *transition* of one state to another exists if, and only if, (1) the program contains a rule which is applicable to the first state, and (2) the effect of firing the updates of this rule at the first state produces the second.

Hence, the definientes of *transition* are *applicability* of a rule to a state, and *effect* of updates. We will consider these notions in turn in the following subsections.

Note that several transitions may emanate from a single state, due to the applicability of more than one rule. This feature introduces non-determinism into evolving algebras on the level of rules. If several rules are applicable at once at some point during a run, one is picked at random.

2.6.1 Applicability of rules

The condition in the **if**-clause of a rule is called its guard. A rule is applicable in a certain state S if its guarding condition evaluates to true in S. If the guard evaluates to false the rule is not applicable. Guards are never allowed to be undefined.

2.6.2 Effects of updates

The effect of an update set upon a state is defined in terms of the effects of all the individual updates that it contains.

Definition The effect of a consistent update set upon a state is the cumulative effect of the individual updates in this set upon the state.

Three questions about this definition need clarification: (1) what does it mean for an update set to be consistent, (2) what are the individual effects of the updates in the update set, and (3) how are these individual effects accumulated to produce the effect of the entire update set?

Individual effects of updates can only be sensibly accumulated if they are *compatible* with each other, i.e. if they do not assign different values to the same subject. An update set whose updates are all compatible with each other is said to be *consistent*. We will answer the first question in section 2.7 by giving a rigorous definition of compatibility of updates and consistency of update sets. In the meantime, it is sufficient to know that the consistency condition of update sets ensures that the effects of the individual updates in these sets can sensibly be accumulated.

We will now successively formulate the answers to the third and second question.

Accumulation of individual effects

Suppose a consistent update set contains the individual updates $u_1, u_2, \ldots, u_n$. Further, suppose state S is the current state upon which the update set is supposed to take effect. We can then construct a sequence of states $S, S_1, S_2, \ldots, S_n$. where each state S_i is the effect upon its predecessor of one of the individual updates $u_1, u_2, \ldots, u_n$ *evaluated in state* S. The last state S_n, then, is the total effect of the update set upon S.

The stipulation that the individual updates be evaluated in state S, rather than in the intermediate states upon which they take effect, is of the utmost importance. Together with the supposition that the update set is consistent, this stipulation ensures that the order of application of the individual updates is of no consequence; the total effect S_n is the same for every application order. Through this irrelevance of application order, parallelism and determinism on the level of update sets is accomplished.

Note, by the way, that the intermediate states $S_1, S_2, \ldots, S_{n-1}$ are *not* nodes of the transition graph. They are auxiliary to the definition of a transition from S to S_n, but they do not actually occur during this transition. In other words, transitions are atomic, even if their update set contains several updates.

Individual effects

We will consider the individual effect of each kind of update in turn. The reader may keep in mind that the local function update is intended to model the phenomenon of destructive assignment, and the extend and remove update are intended to model allocation and deallocation of storage space, respectively.

The effect of a local function update Loosely speaking, the effect of a local function update upon a state is to change the value of the updatable term which is the subject of the update, into the value of the term on the right-hand side. If the updatable term is a variable, this change of value consists in a local change of the interpretation of variables. If the updatable term contains a function identifier, the change of value consists in a local change of the interpretation of this function identifier.

The effect of an extend update Due to the nested update set within an individual extend update, its effect is somewhat more complicated, and can best be described in several steps. Suppose the dynamic sort symbol is d, and the variable is v.

To begin with, the state S upon which the update takes effect is changed in two places. First, the set which is the meaning of d is enlarged with one element. Second, the interpretation of the variable v is changed into this newly added element. We will call the state thus changed S°.

Then, the nested update set of the extend update takes effect upon the state S° in parallel as described in the paragraph 2.6.2. The nested update set is of course not evaluated in S, but in S°, the state upon which it takes effect. We will call the state which results T°.

Finally, the state T° is changed by restoring the variable v to its original value — the value it had in state S. The resulting state T is the total effect of the individual extend update.

In short, the transition $S \xrightarrow{P} T$ is composed as follows:

$$S \longrightarrow S^\circ \xrightarrow{U} T^\circ \longrightarrow T$$

where U denotes the nested update set. Note again that the intermediate transitions and states, S° and T° do not actually occur, are not nodes of the transition graph.

The effect of a remove update A remove update contains a dynamic sort identifier d and an updatable term t. Its effect is to impoverish the set which is the interpretation of d, by one element: the value of the term t.

Consequently, all terms that had the removed element as their meaning (among which t), as well as the terms that have subterms which had the removed element as their meaning, will become undefined.

Since the outermost function identifier of the subject of a local function update, and the sort identifiers of a extend or remove update are always *dynamic* identifiers, updates can never effect a change to the interpretation of static identifiers, but only to that of dynamic identifiers. Hence, the static portions of all states of an evolving algebra are the same.

2.7 Consistency

In this section we will construct a notion of consistency of evolving algebra update sets. We will start in section 2.7.1 by reflecting on the purpose of having a notion of consistency in the theory. Next, we will investigate how much freedom we have in defining the notion, i.e. we will formulate minimal conditions which any definition of consistency must satisfy. Finally, we will construct and adopt a definition.

2.7.1 Purpose of a notion of consistency

The transition rules of an evolving algebra are executed one after an other i.e. sequentially. If several rules are applicable in a state, one of them is non-deterministically chosen to be executed. Thus, at the level of rules, evolving algebras are sequential and non-deterministic.

In contrast, the updates in an update set of an evolving algebra transition rule are executed in parallel, i.e. concurrently or interleaved. Moreover, parallel execution of update sets is supposed to have a deterministic effect. Thus, at the update level, evolving algebras are parallel and deterministic.

However, update sets are not a priori guaranteed to have a deterministic effect under parallel execution. Consider for example the following two updates.

$$a := 1$$
$$a := 2$$

Firing these updates in parallel might either produce a state in which a equals 1 or a state in which a equals 2. Hence, the effect of firing these updates in parallel is not determinate. In section 2.6.2, the effect of firing a set of updates in parallel is defined as the accumulation of the effects of the individual updates in the set. If the order in which the individual effects are accumulated is inconsequential to the total effect, then the parallel execution is determinate.

The purpose of introducing a notion of consistency into the theory of evolving algebras is to provide a condition on update sets which guarantees that the order of the updates is irrelevant, and that consequently the update set has a deterministic effect when fired in parallel. Only consistent update sets will be considered to be executable, and just those evolving algebras will be admitted whose evolution involves firing consistent update sets only.

2.7.2 Sources of inconsistency

The purpose of a notion of consistency puts a restriction on its definition. Consistency must be defined in such a way that it is a sufficient condition for irrelevance of firing order. In this section we will pinpoint pairs of updates whose order is *not* irrelevant. These update pairs must be marked as inconsistent by any notion of consistency, and can be viewed as sources of inconsistency. Through the inspection of these update pairs, we will be guided to the definition of consistency in subsection 2.7.3.

Compatibility

Consistency is a property of update sets. To facilitate its definition, we will reduce it to a property of pairs of updates: compatibility. An update set is said to be consistent in a state S if all individual updates in the set are pair-wise compatible in that state. We will now investigate compatibility of pairs of several kinds of updates.

Local function updates

All definitions of compatibility must exclude pairs of local function updates that assign different values to the same location. The following two updates, for instance, are incompatible:

$$f(a) := 1$$
$$f(a) := 2$$

A stricter definition of compatibility can also be given, which excludes assignments to the same location irrespective of whether the value assigned is the same or different. Such a stricter definition would mark the following two updates incompatible:

$$f(a) := 1$$
$$f(a) := 1$$

The benefit of opting for additional strictness is that it might render evaluation of right-hand sides during consistency checking unnecessary. However, we will adopt the more permissive variant.

Thus, our definition of compatibility will mark two updates $f(t_1 \ldots t_n) := c$ and $g(s_1 \ldots s_n) := d$ incompatible in a state S in the event that three conditions are met:

- The dynamic function names f and g are the same.
- The subterms $s_1 \ldots s_n$ of the first left hand side have the same values in S as the corresponding subterms $t_1 \ldots t_n$ of the second left hand side.
- The right hand sides c and d have a different value in S.

In the special case that the dynamic function names denote 0-ary functions, the updates have the forms $a := c$ and $c := d$. The left-hand sides of these updates have no subterms, which makes the second condition trivially true.

Extend updates

Extend updates contain nested update sets. Any definition of compatibility must entail that an extend update is incompatible with an update that is incompatible to

any of its nested updates. For instance, the following two updates are incompatible:

$f(a) := 1$
new $x : u$ **with**
  $f(x) := 1$
  $f(a) := 2$

The local function update $f(a) := 1$ conflicts with the extend update, because it conflicts with the second nested local function update $f(a) := 2$.

Likewise, the following two updates are incompatible:

new $x : u$ **with**
  $f(a) := 2$
new $x : u$ **with**
  $f(a) := 1$

These two extend updates conflict, because the nested local function updates $f(a) := 2$ and $f(a) := 1$ conflict.

Note, however, that the following two updates do not need to be incompatible:

new $x : u$ **with**
  $f(x) := 2$
new $x : u$ **with**
  $f(x) := 1$

The nested local function updates $f(x) := 2$ and $f(x) := 1$ do not conflict, because the variable x is bound to a different element by each extend update.

So, in general, an extend update **new** $a : v$ **with** $u_1 \dots u_n$ is incompatible to an update u in S if the following condition is met:

- At least one of the updates $u_1 \dots u_n$ is incompatible with the update u in the state S extended with a new element with the name a. (We assume that the name a does not appear in u, so that no name clash occurs.)

Notice that this condition uses the notion of incompatibility which it is intended to help define. This recursiveness does not present a problem, because the number of extend updates to be considered will become smaller at each step.

Remove updates

With respect to the compatibility of remove updates, one can follow a strict or a more permissive line. Consider the following pair of remove updates:

rem $a : v$
rem $b : v$

According to a strict stance towards remove updates, these updates are incompatible when the elements a and b are the same. The motivation for this is that removing the same element twice is not meaningful, or at least very counter-intuitive. However, we could interpret the second removal of an element simply as ineffectual. Then the order in which these removals take effect is inconsequential, and no compelling reason exists to rule them incompatible. Still, we will not adopt this latter, more permissive

stance. We will define the above remove updates to be incompatible if they remove the same element.

We also need to investigate the relationship of remove updates to local function updates. Consider the following pair of updates:

$$\mathbf{rem}\ a : v$$
$$g(s_1 \ldots s_n) := d$$

Two situations can occur in which the compatibility of these updates can be questioned. In the first situation, the element to be removed and one of the subterms of the left hand side are the same. In the second situation, the element to be removed and right hand side are the same. These situations can even occur simultaneously. According to a strict stance, the updates are incompatible in both situations. The motivation for this is that the effect of the local function update is annulled by the remove update. The local function update aims to create a new state in which the left hand side has assumed the right hand side as its value. The remove update, however, causes the left hand side to become undefined. Still, this need not be an absolutely compelling reason to consider the disputed updates incompatible. The remove update annuls the effect of the local function update, irrespective of the order in which they are executed. Hence, firing them in parallel has a determinate effect. Still, we will not adopt this latter, more permissive stance. We will define the above updates to be incompatible when the right hand side or a subterm of the left hand side is the subject of the removal.

2.7.3 Definition of consistency

Having isolated the sources of inconsistency in the previous sections, we will now present a definition of consistency. As promised, this definition will be formulated in terms of compatibility, which is defined here as well.

Consistency

Consistency is a property of update sets with respect to an evolving algebra state. Compatibility is a property of pairs of updates with respect to a state. The former is defined in terms of the latter as follows:

Definition An update set U is consistent in state S if and only if all updates in U are pairwise compatible in S.

Compatibility

We will give a definition of compatibility in terms of *in*-compatibility. In the following table, the first column contains four pairs of potentially incompatible updates. The second column contains for each of these pairs the condition under which incompati-

bility actually occurs.

Pair of updates	Condition of in *in*-compatibility
$f(t_1 \ldots t_n) := c$ $g(s_1 \ldots s_n) := d$	$f \equiv g$ and $\forall i \cdot t_i = s_i$ in S and $c \neq d$ in S
new $a : v$ **with** $u_1 \ldots u_n$ u	$\exists i \cdot u$ and u'_i are incompatible in S'
rem $a : v$ **rem** $b : w$	$a = b$ in S
rem $a : v$ $g(s_1 \ldots s_n) := d$	$\exists i \cdot a = s_i$ in S or $a = d$ in S

In the second condition, S' denotes the state S extended with a new element of v with a name a' that does not occur in u, and u'_i denotes u_i in which a has been renamed to the new name a'. The special cases in which n equals 0 are not explicitly mentioned in this table. Their conditions of incompatibility are straightforward simplifications of the corresponding general cases.

Referring to the given table of update pairs and incompatibility conditions, we can now define compatibility as follows:

Definition A pair of updates is incompatible in a state S if and only if they are of one of the forms in the first row, and they satisfy the corresponding condition in the second row. Otherwise, they are compatible.

All notions in the core theory of evolving algebras have now been explained and defined. Hence, the exposition of the core theory is now complete. In the next chapter, this theory will be enhanced by the introduction of modules.

Chapter 3

Modularization of evolving algebras

3.1 The purpose of modularization 36
3.2 Preparatory adaptations of the theory 36
3.2.1 Start 37
3.2.2 Stop 37
3.3 Modules as parameterized abstractions 37
3.3.1 Categorization of expressions 37
3.3.2 Abstractions 38
3.3.3 Parameterization 38
3.3.4 Return values 40
3.3.5 Visibility across abstraction boundaries 41
3.4 Evolving algebra modules as parameterized abstractions 42
3.4.1 Application of the abstraction principle 42
3.4.2 Application of the correspondence principle 43
3.4.3 Visibility across module boundaries 44
3.5 Functional modules 44
3.5.1 Specification 44
3.5.2 Invocation 45
3.5.3 Semantics 45
3.5.4 Example evolving algebra with functional modules 45
3.6 Procedural modules 46
3.6.1 Specification 47
3.6.2 Invocation 47
3.6.3 Semantics 48
3.6.4 Example evolving algebra with procedural modules 48
3.6.5 More general procedural modules 49
3.7 Modules and consistency 51
3.8 Modularized evolving algebras in the literature 52
3.8.1 Modularization in leanEA 52
3.8.2 Modularization in Diesen 53

3.9 Modules and parallelism 53
3.9.1 Terminological remarks 53
3.9.2 Parallelism in the core theory of evolving algebras 54
3.9.3 Modules and parallelism 55
3.9.4 Parallel and distributed evolving algebras in the literature . 56
3.10 Modules and structured evolving algebra design 57
3.10.1 Modular evolving algebra design 58
3.10.2 Modular evolving algebra proofs 58

In chapter 2 the core theory of evolving algebras was expounded. This chapter will enhance this theory by introducing the concept of a module. We will start by reflecting on the the purpose of modularization of evolving algebras in section 3.1. To prepare the theory for modularization, two small adaptations are made in section 3.2. Section 3.3 expounds a number of concepts from programming languages theory, which views modules as parameterized abstractions. We will take this view as our point of departure for the modularization of evolving algebras in section 3.4. This will lead us to the design of two kinds of evolving algebra modules: functional modules and procedural modules. These will be defined in sections 3.5 and 3.6. The consequences of modularization to the notion of consistency in the theory of evolving algebras are investigated in 3.7. We will conclude the chapter by investigating the benefits of functional and procedural modules with for parallelism and structured evolving algebra design.

3.1 The purpose of modularization

The purpose of modularizing evolving algebras is twofold.

One purpose of modularizing evolving algebras is to be able to divide an evolving algebra specification into relatively independent parts. This relative independence will make it simpler to formulate, understand, adapt, and compare specifications. Especially, large evolving algebras stand in need of modularization. In section 3.10 it will be shown that modules can be used in structured evolving algebra design.

Another purpose of modularizing evolving algebras is to incorporate large-grain parallelism into the language. In the core theory of evolving algebras, updates are executed in parallel, and expressions may be evaluated in parallel. But due to the limited complexity of updates and expressions, the granularity of this parallelism is small. Modularizing evolving algebras can remove the restrictions of complexity of updates, thereby making the granularity of update parallelism variable. In section 3.9 the use of modules in parallel evolving algebras is explained.

3.2 Preparatory adaptations of the theory

In this section two small adaptations of the simple theory are proposed which will make the introduction of modules into the theory smoother.

3.2.1 Start

In the simple theory, both the fixed meanings of the static identifiers of an evolving algebra, and the initial meanings of the dynamic identifiers are specified by giving the initial state. In the adapted theory, we will separate these concerns. The fixation of the static identifiers will be done by giving the static portion of the initial state. The initialization of the dynamic identifiers is done by specifying a distinguished set of updates, called the *start* update set. This update set will be fired at the initial state before the execution of the program. In the example given in the informal introductory chapter (chapter 1) this method of two-phase initial state description already been demonstrated.

3.2.2 Stop

The execution of an evolving algebra program terminates when a state is reached in which no rules are applicable. In the simple theory, no distinction is made between abnormal and normal termination. In the adapted theory, we will be able to make this distinction by providing a *stop* condition. When the execution of the program terminates, the termination is said to be normal if and only if the stop condition evaluates to true in the final state.

Stop conditions are useful for documentation and detection of unintended behavior of evolving algebras. They can also be helpful in the construction of modular proofs about modular evolving algebras.

3.3 Modules as parameterized abstractions

The construction of a theory of modular evolving algebras is not much unlike the construction of a structured programming language. For this reason we will take a number of principles of programming language theory into consideration. This section presents the relevant principles, while the next section applies these principles to the language of evolving algebras.

3.3.1 Categorization of expressions

Traditionally, at least two categories are distinguished among the phrases of programming languages. The first category comprises those phrases which are *evaluated*, called functional expressions, or just expressions. The second category consists of those phrases which are *executed*, called commands, statements or imperative expressions[1]. The essential difference between the two categories is the following: the first kind of expression represents a computation which produces a value, the second kind of expression represents a computation which alters a state.

The expressions of evolving algebras can be categorized in this way as well. Evolving algebra terms fall in the category of functional expressions; evolving algebra updates fall in the category of imperative expressions.

[1] These two categories are not disjunct in all languages. In fact, in languages such as Algol-68 and ML these two categories coincide.

3.3.2 Abstractions

In programming language theory, the word "abstraction" is used to refer to programming language constructs that enable the programmer to separate the *specification* of a sub-computation from its *use* or *invocation*. Abstractions are categorized by the expression category to which their invocation belongs. Two common categories of abstractions are function abstractions, which are abstractions over functional expressions, and procedure abstractions, which are abstractions over imperative expressions. This correspondence between categories of abstractions and categories of expressions, is expressed by the following principle of programming language theory ([Wat90, 92]):

Abstraction principle It is possible to construct abstractions over any category of expressions[2], provided only that the phrases of that category specify some kind of computation.

In the next section we will apply this principle to the categorization of expressions of evolving algebra theory.

The separation of specification and invocation accomplished by abstraction creates the need of passing information from invocation to specification and *vice versa*. In the following subsections, we will discuss three available methods for passing information across the boundary of an abstraction: parameters, return expressions, and identifiers that are visible across abstraction barriers.

3.3.3 Parameterization

An abstraction does not need to represent one particular computation, but may be parameterized, thus representing a family of computations. The parameters used in the specification of the abstraction are called *formal* parameters. The parameters supplied at the invocation of the abstraction are called *actual* parameters.

The formal parameters of a parameterized abstraction must be associated to the actual parameters of a particular invocation of the abstraction. The manner in which this association is realized is called a parameter mechanism. One group of parameter mechanisms are called copy mechanisms, because they are most naturally implemented by copying. Another group of parameters are called denotational mechanisms, because they are usually implemented by referencing. We will discuss these two groups in turn.

Copy parameter mechanisms

Three parameter mechanisms fall into the category of copy mechanisms. These mechanisms are distinguished by the moments at which the value of the actual parameter is transferred to the formal parameter and *vice versa*.

value parameters At the moment of invocation, the value of the actual parameter is transferred to the formal parameter. Value parameters can be used to pass information *into* an abstraction only.

result parameters At the moment of return, the value of the formal parameter is transferred to the actual parameter. Return parameters can be used to pass information *out of* an abstraction only.

[2] In Watt [Wat90] the term "syntactic class" is used in stead of category of expressions.

value-result parameters The *value-result parameter* is a combination of the other two. At the moment of invocation the value of the actual parameter is transferred to the formal parameter, and at the moment of return the value of the formal parameter is transferred to the actual parameter. Value-result parameters can be used to pass information both into and out of an abstraction.

Assignments are allowed on each of these kinds of parameters, but an assignment to a formal parameter has no effect on the value of a another formal parameter that corresponds to the same actual parameter.

This group of parameter mechanisms are called copy mechanisms, because the transfer of values of actual parameters to formal parameters and *vice versa* is most naturally implemented by a copy operation.

Denotational parameter mechanism

Two parameter mechanisms fall into the category of denotational mechanisms. The distinction between these parameters concerns whether assignments are allowed to be performed on them or not.

constant parameters Assignments on constant parameters are forbidden.

variable parameters Assignments on variable parameters are permitted. An assignment to a formal parameter changes the values of all formal parameters that correspond to the same actual parameter.

In case of both these kinds of parameters, the value of the actual parameter is transferred to the formal parameter at the moment of invocation. Hence, both are usable to pass information into an abstraction. Only in the case of variable parameters are the values of formal parameters transferred to their actual parameters, at the moment of return. In case of a constant parameter, the actual parameter does not obtain the value of the formal parameter. As a result, constant parameters are not suited to pass information out of an abstraction, while variable parameters are.

These two parameter mechanisms are called denotational mechanisms, because they are usually implemented by referencing. At the moment of invocation, a reference is installed from the formal parameter to the actual parameter, and inspections and assignments of the formal parameter are performed on the referenced actual parameter. As a consequence, at the moment of return the values of actual variable parameters are already equal to the values of the formal parameters, and no explicit transfer of values is needed.

Comparisons can be made between similar parameter mechanisms from different groups. For instance, value and constant parameters are similar in that they are used to transfer information into, but not out of an abstraction. They are different in that assignment is allowed on a value parameter, but not on a constant parameter.

Value-result and variable parameters are similar in that they are used to transfer information both into and out of the abstraction. They are different in that an assignment to a formal variable parameter influences all formal variable parameters that share its actual parameter, while assignments to formal value-result parameters never influences the value of any other formal parameter.

When two formal parameters correspond to the same actual parameter, they are said to be *aliases* of each other. The occurrence of *aliasing* is often not intended by the programmer, and can be the cause of serious flaws in the program. In the case

of aliased value-result parameters, different values are transferred to the same actual parameter. Which value overrules which is decisive for the result of the invocation, but the programmer is often unaware of the conflict. In the case of aliased variable parameters, an assignment to one variable changes the value of both. The programmer usually does not anticipate this. In order to prevent mistakes due to unanticipated aliasing, a programming language designer might choose to prohibit aliasing altogether. If such a prohibition is in effect, the semantical difference between value-result parameters and variable parameters vanishes.

The properties of the parameter mechanisms from both groups are summarized in the following table:

	value	result	value-result	constant	variable
Assignment allowed	yes	yes	yes	no	yes
Assignment changes alias	no	no	no		yes
Pass info into abstr.	yes	no	yes	yes	yes
Pass info out of abstr.	no	yes	yes	no	yes

The second line of this table indicates for each parameter mechanism whether assignment to a formal parameter changes the values of its aliases inside the abstraction. For constant parameters, this line of the table has no entry. This is due to the fact that assignment is not allowed on constant parameters.

The correspondence principle

There is a certain similarity between formal parameters and declarations. Both introduce a new name into the vocabulary of the program (both contain a declarative (vs. applied) occurrence of a name). The difference between a formal parameter and a declaration, is that a declaration not only specifies the new name, but also the entity to which it will be bound. A formal parameter, on the other hand, only specifies the new name. The entity to be bound is specified by an actual parameter in an invocation.

The correspondence between declarations and formal parameters is expressed by the following principle of programming language theory ([Wat90, 99]):

Correspondence principle For each form of declaration there exists a corresponding parameter mechanism, and *vice versa*.

The correspondence principle should be interpreted to be prescriptive as well as descriptive. The *regularity* of a language benefits if a correspondence exists between its forms of declaration and its kind of parameters. In particular, no parameter mechanisms should be introduced into a language that do not correspond to one of its forms of declaration.

When several parameter mechanisms are available in a language, it is imperative that for each parameter it is evident according to which mechanism it operates. In many languages, the kinds of the parameters are immediately evident only in the specification of the abstraction. The kinds of the actual parameters in the invocation must usually be derived by looking up the kind of the corresponding formal parameter in the specification. Most likely, the readability of programs improves when the kind of actual parameters can be gleaned from the invocation itself.

3.3.4 Return values

The parameter mechanisms described above serve to pass values into and out of abstractions. But there is another way of passing a value out of an abstraction, which

makes no use of parameters. This method is used in case of abstractions over functional expressions, and is generally known as *returning* a value. In this mechanism, no actual parameter is present, since the invocation itself takes on the return value. No formal parameter needs to be present either, since inside the abstraction the value can be produced by evaluating a functional expression in stead of by assigning a value to a variable. In fact, the following mechanisms are most common:

- A *return expression* is explicitly specified for each abstraction, which is evaluated at the moment of exiting the abstraction. The value thus obtained is the return value of the abstraction (Modula2, Pascal).
- The body of the abstraction is itself a functional expression. The value obtained by evaluating it is at the same time the return value of the abstraction (functional languages).

Which of these two mechanisms is most appropriate is dependent on to which category of expressions the body of the abstraction belongs.

All abstraction over categories of expressions which are evaluated, must return values according to some return mechanism.

Return principle Any abstraction over a category of expressions which can be evaluated, must return a value.

The abstraction principle and the correspondence principle are merely guidelines for the language designer, which lead to increased regularity. But the return principle can not be dismissed: its satisfaction is compulsory.

3.3.5 Visibility across abstraction boundaries

Parameters and return expressions are used to pass information across abstraction boundaries. A third method exists to pass information into and out of abstractions. This method amounts to allowing identifiers declared outside the abstraction to be visible inside the abstraction. In subsection 3.3.3 we assumed and required all identifiers appearing free in an abstraction body to be bound by parameter declarations in the abstraction header. It is possible to drop this requirement, and to allow free identifiers to appear in an abstraction that are not bound by parameter declarations in the header, but by declarations outside the abstraction. These identifiers can be viewed as non-locally declared parameters.

Allowing visibility across abstraction barriers creates the obligation of determining which declaration binds each free identifier. Two different binding strategies can be adopted: dynamic binding or static binding. According to the strategy of dynamic binding, the context of *invocation* of the abstraction determines by which declarations its free identifiers are bound. According to the strategy of static binding, the context of *specification* of the abstraction is determinative. Static binding has several advantages over dynamic binding. Firstly, static binding can be done at compile-time, while dynamic binding can only be performed at run time. Thus, binding errors are detected in a later phase. Secondly, dynamic binding leads to programs that are hard to understand because the binding declarations can not be determined from the program text alone. Finally, and most importantly, dynamic binding is incompatible with static type checking These three reasons lead us to favor static binding over dynamic binding.

Since each identifier that is visible across an abstraction barrier can be viewed as a non-locally declared parameter, it must be one of the five kinds of parameter listed in subsection 3.3.3. Recall that result parameters, value-result parameters and variable parameters are suited to pass information *out of* an abstraction, while value parameters and constant parameters are suited to pass information *into* the abstraction only. To use free identifiers for the purpose of passing information *out of* an abstraction yields programs that very hard to understand. In these programs one cannot glean from the invocations of abstractions nor from their headers, which variables are affected by them. As a result, well-designed programming languages will not permit free identifiers that operate as result parameters, value-result parameters or variable parameters. Hence, free identifiers should either be non-locally declared constant parameters or non-locally declared value parameters.

The possibility of visibility across abstraction boundaries permits parameter lists to be shortened considerably. As a result, programs can become more concise, and therefore easier to understand. On the other hand, visibility across abstraction boundaries has as disadvantage that abstraction specifications become dependent upon their context, and can be understood only as parts of the program in which they appear.

3.4 Evolving algebra modules as parameterized abstractions

Bearing in mind the concepts and principles of programming language theory presented in the previous section, we will set out to modularize the theory of evolving algebras. In this section we will consider the consequences of the concepts and principles of programming language theory to the theory of evolving algebras. In the following sections, we will present functional and procedural modules.

3.4.1 Application of the abstraction principle

There are two categories of evolving algebra phrases which specify computations: *terms* and *updates*. Consequently, the abstraction principle informs us that it is possible to construct abstractions over terms and abstractions over updates. Since terms specify functional computations, and updates specify imperative computations, we will call these abstractions *functional modules* and *procedural modules*, respectively.

It is clear that procedural modules must be callable as updates, and functional modules as terms. There is a choice, however, as to what category of expressions is encapsulated by each kind of module, i.e. what kind of expressions the module bodies are. One possibility is to let a procedural module encapsulate an update, and a functional module a term. However, since the evolving algebra language does not include local declarations, nor conditional terms or updates, the resulting expressiveness of module bodies would be very limited. Therefore, our approach will be to let both kinds of modules encapsulate an entire evolving algebra. Consequently, the full expressiveness of the evolving algebra language will be available within each module.

To build functional and procedural modules from the evolving algebras they are supposed to encapsulate, we need to provide them with functional and procedural interfaces, respectively. Sections 3.5 and 3.6 will explain how this can be done.

3.4.2 Application of the correspondence principle

Evolving algebra specifications can contain four kinds of declaration:

static sort declaration Specifies the name of a static sort and the mathematical set to which it is to be interpreted.

static function declaration Specifies the name of a static function and the mathematical function to which it is to be interpreted.

dynamic sort declaration Specifies the name of a dynamic sort, which will initially be interpreted as an empty mathematical set.

dynamic function declaration Specifies the name of a dynamic function, which will initially be interpreted as a function that is undefined for all its arguments.

Dynamic sorts and functions can be subjects of updates. Static sorts and functions can not be updated.

According to the correspondence principle, parameter mechanisms can be conceived corresponding to each of these kinds of declaration.

static sort parameter Upon entering the abstraction a static sort parameter receives the value of the actual parameter as its interpretation. No extend or remove updates are allowed on the formal parameter. Thus, static sort parameters are instances of *constant parameters*.

static function parameters Upon entering the abstraction a static function parameter receives the value of the actual parameter as its interpretation. No local function update is allowed on the formal parameter. Thus, static function parameters are instances of *constant parameters*.

dynamic sort parameters Upon entering the abstraction a dynamic sort parameter receives the value of the actual parameter as its interpretation. New-updates and remove-updates are allowed on the formal parameter. Since aliasing will be prohibited, there is no semantic distinction between value-result parameters and variable parameters. Thus, dynamic sort parameters are equally instances of *value-result parameters* as they are instances of *variable parameters*.

dynamic function parameters Upon entering the abstraction a dynamic function parameter receives the value of the actual parameter as its interpretation. Local function updates are allowed on the formal parameter. Again, aliasing is prohibited. Thus, dynamic sort parameters are instances of *value-result* or *variable parameters*.

Not all of these parameter mechanisms are equally suitable to be introduced into the theory of evolving algebras. In particular, the precipitate introduction of static sort parameters and static function parameters would entail more radical departures from the core theory than modularization calls for. Static sort parameters give rise to type polymorphism, and static function parameters give rise to higher order functions. Both these features are foreign to the core theory, and are not required for modularization. Therefore, we ban static sort parameters altogether, and restrict static function parameters to 0-ary functions names.

The remaining kinds of parameter are not equally suitable for both functional and procedural modules. We will need to decide for each kind of module which kinds of parameter to allow.

3.4.3 Visibility across module boundaries

If visibility across module boundaries with static binding is to be allowed in the modularized theory of evolving algebras, then it must be possible to specify modules in the context of other modules. Hence, nested specifications is a prerequisite to allowing names to be visible across module bodies.

Our proposal for modularizing the theory of evolving algebras will not involve nested specifications, and consequently, no visibility across abstraction boundaries will be possible.

If nested specifications and visibility across module boundaries were introduced into the theory in a later stage, it would need to be decided whether free identifiers are to be handled as value parameters or as constant parameters. Names that are declared outside a module as *static* names, are preferably handled as constant parameters. If they were handled as value parameters, they would sudden become dynamic inside the module.

In the following subsections we will explain for both kinds of modules how they are *specified*, how they are *invoked*, and what their semantics are.

3.5 Functional modules

To make a functional module out of an evolving algebra, it must be provided with a functional interface, i.e. it must be made callable as a function, it must take arguments and return a result. Since terms and updates are completely disjunct categories, functional modules must be purely functional, i.e. side-effects must be prohibited. Consequently, neither dynamic sort parameters, nor dynamic function parameters are allowed, since they are instances of value-result or variable parameters. Thus, the parameters of a functional module must static function parameters of arity zero.

Functional modules are abstractions over terms and terms are functional expressions. Thus, according to the return principle, a functional module must return a value. The body of a functional module is an evolving algebra, which is not a functional expression. Hence, the second return mechanism, which uses the value of the body as return value, is not an option in this case. We will use the first return mechanism, which demands a return expression to be explicitly specified.

3.5.1 Specification

An evolving algebra can be provided with a functional interface by attaching a suitable header to its specification. We propose the following general form for this header:

$$\textbf{module}\ func\ (\textbf{sf}\ b_1 : B_1, .., b_n : B_n)\ r : R$$
$$\langle body \rangle$$

where

- $func$ is the name of the module.
- $b_1, .., b_n$ are function names.
- $B_1, .., B_n$ are sort names
- r is a term.

- R is a sort name.
- The body is a complete evolving algebra specification.

The sort names $B_1, .., B_n$ and R, should be declared in the signature of the body. Together with the names $b_1, .., b_n$ the signature of the body forms the signature of the module. The expression r, as well as all expressions occurring in the body should be built from the names in this module signature. The names $b_1, .., b_n$ are allowed to occur in the body only as static function names, not as dynamic ones.

3.5.2 Invocation

The header of a functional module specification ensures that it can be called as a function by other modules. In the calling module, the name of the called module is used as a *static* function name. Thus, it may be used to built any expression which is not used as the subject of an update. The general form of the invocation is as follows:

$$func(b_1, .., b_n)$$

where

- $b_1, .., b_n$ are terms.

The terms $b_1, .., b_n$ can be built from both static and dynamic function names, even though their corresponding formal parameters can only be used as static function names.

3.5.3 Semantics

The parameters of the functional module are all constant parameters. Hence, at the moment of invocation the formal parameters are bound to the values of their actual parameters. Then, the body, which is a complete evolving algebra, is executed. When this evolving algebra terminates — if it terminates — the return expressions is evaluated in the final state. The value obtained is returned as the value of the module invocation.

3.5.4 Example evolving algebra with functional modules

We will now present an example of a modularized evolving algebra which make use of functional modules. This modularized evolving algebra consists of two functional modules. The first functional module implements multiplication of two natural numbers. Its specification is:

module mult (**sf** $n, m : N$) $r : N$

Signature:

static sorts
$N \Rrightarrow \mathbb{N}$
static functions
$+ : N \times N \rightarrow N \quad \Rrightarrow \quad + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
$- : N \times N \rightarrow N \quad \Rrightarrow \quad - : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
dynamic functions
$r : N$
$i : N$

Start:

$i := n$
$r := 0$

Program:

if $i \neq 0$
then $r := r + m$
$\quad i := i - 1$

Stop: $i = 0$

This module works by letting a counter run from n to zero, and adding m to an accumulator at each step.

The second module of our example evolving algebra implements the factorial function. It invokes the *mult* module to perform multiplication.

module fac (**sf** $n : N$) $r : N$

Signature:

static sorts
$N \Rrightarrow \mathbb{N}$
static functions
$- : N \times N \to N \Rrightarrow - : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
dynamic functions
$r : N$
$i : N$

Start:

$i := n$
$r := 1$

Program:

if $i \neq 0$
then $r := mult(r, i)$
$\quad i := i - 1$

Stop: $i = 0$

This module operates by letting a counter run from n to zero, and multiplying an accumulator by this counter at each step.

3.6 Procedural modules

To make a procedural module out of an evolving algebra, it must be provided with a procedural interface, i.e. it must be made callable as an update, it must take arguments and change the interpretation of dynamic sort names or dynamic function names. Like regular updates, invocations of procedural interfaces have side-effects upon the state at which they are fired. Therefore, procedurale modules must have dynamic sort and function parameters as well as static function parameters.

3.6.1 Specification

An evolving algebra can be provided with a procedural interface by attaching a suitable header to its specification. We propose the following general form for this header:

module *proc* (**sf** $b_1 : B_1, .., b_l : B_l$; **ds** $C_1, .., C_m$; **df** $d_1 : D_1, .., d_n : D_n$)
$\langle body \rangle$

where

- *proc* is the name of the module.
- $b_1, .., b_l$ are function names.
- $B_1, .., B_l$ are sort names.
- $C_1, .., C_m$ are sort names.
- $d_1, .., d_n$ are function names.
- $D_1, .., D_n$ are sort names or sort expressions of the form $(S_1 \ldots S_m) \rightarrow S$.
- The body is a complete evolving algebra specification.

The module signature consists of the body signature extended with the parameter declarations. All expressions in the body should be built from the names in the module signature. The sort names $B_1, .., B_l$ and the sort names (appearing in) $D_1, .., D_n$ should be declared in the module signature. The names $b_1, .., b_n$ are allowed to occur in the body as static function names. The names $d_1, \ldots, d_n$ can be used as dynamic function names. The sort names $C_1, .., C_m$ can be used as dynamic sort names in the body.

3.6.2 Invocation

The procedural interface ensures that the module can be called as an update by other modules. The general form of the invocation of a procedural module is:

$$(C_1, .., C_m;\ d_1, .., d_n) := proc(b_1, .., b_l)$$

where

- $b_1, .., b_l$ are terms.
- $C_1, .., C_m$ are dynamic sort names.
- $d_1, .., d_n$ are dynamic function names.

The names $C_1, .., C_m$ and $d_1, .., d_n$ *must* be dynamic. The terms $b_1 .., b_l$ can be built from both dynamic and static function names. To prevent aliasing, the dynamic sort names $C_1, .., C_m$ and the dynamic function names $d_1, .., d_n$ are all required to be distinct.

3.6.3 Semantics

The first set of parameters of the procedural module are constant parameters. The other two sets of parameters are value-result or variable parameters. At the moment of invocation all the formal parameters are bound to the values of their actual parameters. Then, the body, which is a complete evolving algebra, is executed. When this evolving algebra terminates — if it terminates — control returns to the calling module. If the last two sets of parameters are value-result parameters, their values in the final state are bound to the actual parameters just before return of control.

3.6.4 Example evolving algebra with procedural modules

We will now present a modularized evolving algebra which contains procedural modules. The first module creates a linked of natural numbers from zero to n.

module count (**sf** $n : N$
ds $ListElem$
df $head : ListElem \to N$,
$tail : ListElem \to ListElem$,
$root : ListElem$)

Signature:

static sorts
$N \Rrightarrow \mathbb{N}$
static functions
$+ : N \times N \to N \Rrightarrow + : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
dynamic functions
$last : ListElem$
$i : N$

Start:

new $e : ListElem$ **with**
$head(e) := 1$
$root := e$
$last := e$

Program:

if $head(last) \neq n$
then **new** $e : ListElem$ **with**
$head(e) := head(last) + 1$
$last := e$
$tail(last) := e$

Stop: $head(last) = n$

The second module uses the first. It creates a list of factorials, just like the example in section 1.2. It operates by invoking the first module to create a list of increasing natural numbers. Then it tranverses this list and multiplies each number with its predecessor, which has already been multiplied.

module faclist (**sf** $n : N$
ds $ListElem$
df $head : ListElem \rightarrow N,$
$tail : ListElem \rightarrow ListElem,$
$root : ListElem$)

Signature:

static sorts
$N \Rrightarrow \mathbb{N}$

static functions
$* : N \times N \rightarrow N \Rrightarrow \cdot : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

dynamic functions
$current : ListElem$

Start:

$(ListElem; head, tail, root) := count(n)$

Program:

initialize
if $\neg defined(current)$
then $current := root$
step
if $defined(current) \wedge defined(tail(current))$
then $head(tail(current)) := head(tail(current)) * head(current)$
$current := tail(current)$

Stop: $\neg defined(tail(current))$

3.6.5 More general procedural modules

The proposal of procedural modules just presented can be generalized with respect to the actual parameters that are allowed in module invocations. Procedural module invocation are of the following form:

$$(C_1, .., C_m;\ d_1, .., d_n) := proc(b_1, .., b_l)$$

According to subsection 3.6.2, the actual parameters $d_1, .., d_n$ are required to be dynamic function names. We will show by a schematic example that this requirement can be relaxed, so that more general procedural module invocations are possible.

We will sketch a situation in which a procedural module *proc* is invoked by an evolving algebra. Assume that the signature of the evolving algebra contains at least the following declarations:

dynamic functions
$a : A$
$b : B$
$c : C$
$f : A \rightarrow C$
$g : A \times B \rightarrow C$

Further, suppose the procedural module *proc* has the following header:

module *proc* (**df** $f' : A' \rightarrow C', c' : C'$)

In the body of the module, the following updates might occur:

$$f'(x) := t$$
$$c := s$$

where x, t and s are terms of appropriate sorts.

Given the signature and the procedural module header, the following invocation of the module *proc* is possible:

$$(f, c) := proc$$

When this invocation is executed, the two example updates inside the module will have the effect of the following two updates outside the module:

$$f(x) := t$$
$$c := s$$

In the invocation just shown, both f and c are dynamic function names. In the position of the 0-ary dynamic function name c of sort C, we could also allow an *updatable term* of sort C. For instance, the updatable term $f(a)$ is of appropriate type, and can be used as follows:

$$(f, f(a)) := proc$$

When this invocation of *proc* is executed, the example updates will have the effect outside the module of the following updates:

$$f(x) := t$$
$$f(a) := s$$

Thus, as actual parameters for 0-ary dynamic function parameters, we can allow not only dynamic function names, but updatable terms in general.

Even more generality can be obtained. Let the partial applications of the binary function g of type $A \times B \to C$ be denoted by $g(a, _)$ and $g(_, b)$ of types $B \to C$ and $A \to C$ respectively. We can than formulate the following invocation of the procedural module *proc*:

$$(g(_, b), f(a)) := proc$$

When this invocation is executed, the example updates inside the module have the same effect as the following updates outside the module:

$$g(x, b) := t$$
$$f(a) := s$$

Thus, inside the module, the partially application of g is completed to a full application by supplying the argument x.

So now we have three kinds of actual parameters: dynamic function names, updatable terms, and partially applied dynamic function names. These three possibilities actually form a continuum. A dynamic function name g can be viewed as a completely unapplied function name: $g(_, _)$. An updatable term can be viewed as a completely applied function name: $g(a, b)$. In between these extremes, we can position the partially applied dynamic function names $g(a, _)$ and $g(_, b)$.

3.7 Modules and consistency

The introduction of functional modules into the theory of evolving algebra has no consequences for consistency. This is due to the fact that functional modules are purely functional, and their invocations are terms, not updates. The extension of the theory of evolving algebras with procedural modules, on the other hand, implies the introduction of a new kind of update: the procedural module invocation. Consequently, the definitions of consistency of update sets and compatibility of updates must be extended to accommodate this new kind of update.

As was the case for consistency in the core theory, two basic strategies can be adopted towards consistency for modules: a permissive strategy and a strict strategy.

The permissive strategy treats the updates encapsulated within the procedural module in a way similar to the updates nested within an extend update. A procedural module invocation can then be defined to be compatible with another update, if all its encapsulated updates are compatible with this other update. As an example, recall the procedural module *proc* of the previous section, and its encapsulated updates:

$$\begin{array}{l} \textbf{module}\ proc\ (\textbf{df}\ f' : A' \to C', c' : C') \\ \quad\quad \vdots \\ f'(x) := t \\ c := s \\ \quad\quad \vdots \end{array}$$

Given this module definition, consider the following pair of updates:

$$\begin{array}{l} (f, a) := proc \\ f(y) := z \end{array}$$

These updates are incompatible if y evaluates to the same value as x, while z and t evaluate to different values.

The notion of consistency obtained through the permissive strategy rules out as few update sets as possible. However, the consistency checking algorithm for this notion is extremely laborious. It involves executing the module in its entirety in order to establish which updates it performs. Since the body of the module can be an entire evolving algebra, there is no limit to the amount of work that consistency checking entails.

The alternative to the permissive strategy is the strict strategy. The strict strategy does not look at the updates encapsulated by a module, but only at the dynamic function names on the left-hand side of its invocation. A procedural module can then be defined to be compatible with another update, if none of the dynamic function names in the left-hand side of the invocation appear as subject in this update. So, for instance, the updates in each of the following two pairs are incompatible:

$$\begin{array}{ll} (f, a) := proc \quad\quad & (f, a) := proc \\ a := x & f(y) := z \end{array}$$

Strict consistency implies permissive consistency. If a dynamic function name f is not contained in the left hand side of the invocation of *proc*, then the execution of *proc* can not encapsulate an update which changes f. Therefore, the invocation can not be incompatible with an update of which f is the subject.

So far we have only considered local function updates. We must now deal with remove updates. Consider the following header of a procedural module *proc'*

$$\textbf{module } proc' \ (\textbf{ds } D', \textbf{df } x : D')$$

The body of this module may contain remove updates and extend updates on D'. Assume invocations of this procedural modules occur in the following pairs of updates:

$$\begin{array}{ll} (D; a) := proc' & (D; a) := proc' \\ \textbf{rem } a : D & f(x) := y \end{array}$$

According to the permissive strategy towards consistency of modules, these pairs may or may not be compatible, depending on what updates are actually performed by the procedural module invocations. To check consistency according to this strategy involves running the encapsulated evolving algebras.

A strict stance towards consistency of modules can avoid running encapsulated evolving algebras, if it is combined with the permissive stance towards remove updates in the core theory. According to this permissive stance, remove updates are always compatible with each other and with other updates. As a consequence, the two example update pairs just mentioned are always compatible, irrespective of the updates performed by their bodies. Thus, with respect to remove updates, the strict strategy towards module consistency is actually more permissive than the permissive strategy, due to the presupposed permissiveness towards remove updates in the core theory.

The strict notion of consistency can also be extrapolated to general procedural modules. For instance, the following updates are incompatible:

$$\begin{array}{l} (g(1, _), a) := proc \\ g(1, x) := y \end{array}$$

This incompatibility is due to the fact that the subject of the second update $g(1, x)$ is a completion of the partially applied function $g(1, _)$, which appears as an actual parameter in the first update.

The strict strategy make consistency checking easier, at the cost of disqualifying update sets that would have been retained by the permissive strategy. Adoption of the first strategy makes it necessary to execute the module invocations in order to check consistency. After all, the bodies of modules are complete evolving algebras, which makes it rather laborious to assess which local function updates will be performed by it.

3.8 Modularized evolving algebras in the literature

3.8.1 Modularization in leanEA

In [BP95] Beckert and Posegga describe an evolving algebra tool called leanEA. This tool accepts an evolving algebra language which includes modules. The non-modular core of this evolving algebra language is considerably less rich than the core theory described in chapter 2. In particular, it does not support dynamic sorts, and has no remove or extend updates. Also, static and dynamic functions are not strictly distinguished. The modules supported by leanEA's evolving algebra language are functional modules. These functional modules are very similar to the functional modules described in section 3.5. Three differences must be noted. Firstly, the parameters of leanEA's functional modules are value parameters instead of constant parameters. Secondly, the return values of leanEA's modules are not single values, but lists of values. Thirdly, the functional modules are not invoked directly in an evolving algebra,

but by mediation of a Prolog predicate. There is no concept of procedural modules in leanEA.

3.8.2 Modularization in Diesen

In [Die95b] Diesen presents a modularization of evolving algebras which deviates radically from the modularization proposed in this chapter. Diesen's modules are not parameterized abstractions, but rather collections of rules operating in a single name space. This name space is partially visible to each module. Information is not passed between modules through parameters, but by updating and inspecting shared names. The modules pass control to each other by jumps. Modules that return control to the module that gave them control initially are called sub-routines. Modules that pass control on to other modules are called co-routines.

3.9 Modules and parallelism

3.9.1 Terminological remarks

Before we will embark on a discussion of parallelism in the theory of evolving algebras, it is necessary to clarify the terminology that will be used. The distinctions between the terms 'parallel', 'concurrent', 'interleaved', and 'distributed' as used in this section will be explained.

In common usage, the terms 'parallel' and 'distributed' do not have neatly defined meanings. Both terms denote processes which consist of subprocesses that are executed, or could be executed, simultaneously. Commonly, the term 'distributed' is used to indicate that the subprocesses are relatively independent, and are executed on fairly autonomous physical machines. Occasionally, the term 'parallel' is used in contrast with 'distributed'. More often, parallel processes are considered to subsume distributed processes.

In this chapter we will use the terms 'parallel' and 'distributed' not in their colloquial vague meanings, but to make a strict semantical distinction. We will define the terms in turn.

Parallel processes

In the case of a parallel process, the subprocesses are ordered in a tree-shaped hierarchy. A parent-child relationship exists between processes at different levels in the hierarchy. Siblings do not communicate with each other. They are not dependent on each other's results. No synchronization between siblings is needed. All communication takes place between parents and children. Each parent depends on the results of its child processes.

Distributed processes

In the case of distributed processes, a hierarchy is neither presupposed, nor precluded. The subprocesses are dependent on each other's results. They communicate with each other by passing messages, or by writing data into a shared memory space, which can be inspected by one or more of the others.

Concurrent and interleaved computations

The subprocesses of a parallel of distributed proces can be executed at the same time on different processors, or at different times on the same processor. In the former case, the subprocesses are said to be executed concurrently, in the latter case their execution is said to be interleaved. Thus, the distinction between interleaved and concurrent computations concerns the allocation of processor time. We are not concerned not with resource allocation in the present discussion. As a consequence, we will not discuss issues of concurrency and interleaving.

3.9.2 Parallelism in the core theory of evolving algebras

Parallelism is present in the core theory of evolving algebras at several places. These occurrences of parallelism fall into two general categories: parallel evaluation, and parallel execution. We will discuss the occurrences of parallelism in turn.

Parallel evaluation of expressions

There are two kinds of expressions in evolving algebras: terms and conditions. In general, terms have the following form:

$$f(t_1, \ldots, t_n)$$

The evaluation of this term demands the evaluation of its subterms $t_1, \ldots, t_n$. These subterms can be evaluated in parallel. Hence, when the evaluation of the complete term is the parent process, the evaluation of its subterms are the child processes.

Conditions may have one of the following general forms:

$$c_1 \oplus \ldots \oplus c_n$$
$$t_1 = t_2$$

where *oplus* denotes a binary logical operation. The evaluation of a condition of the first form demands the evaluation of its subconditions $c_1, \ldots . c_n$. These subconditions can be evaluated in parallel, by child processes of the process that evaluates the entire condition. Likewise, the evaluation of t_1 and t_2 can be performed by parallel child processes of the evaluation of a condition of the second form.

So, in general, when the evaluation of an expression requires the evaluation of subexpressions, these subexpressions can be evaluated in parallel.

Parallel evaluation of left and right hand sides

Local function updates have the following general form:

$$f(t_1, \ldots, t_n) := t$$

The execution of such an update requires the evaluation of the right hand side t, as well as the evaluation of the subterms $t_1, \ldots, t_n$ of the updatable terms at the left hand side. All these evaluations can be performed in parallel. In this case, the parallel evaluation processes are child processes of the execution of the update.

Parallel execution of updates

As was mentioned in earlier chapters (e.g. section 2.7.1), evolving algebras support parallelism at the update level. The updates in the update set of a transition rule are executed in parallel. If any of these updates contain nested updates, these are executed in parallel as well. Consider, for instance, the following set of updates:

$a := 1$
new $x : D$ **with**
  $x := 2$
  $f(x) := 3$
$f(b) := g(4,5)$

All four local function updates that occur nested or not nested in this update set, are executed in parallel. The parallel processes that perform these individual updates operate upon a single name space. The consistency condition imposed on update sets ensures that no conflicts can arise due to this sharing. Each update in a consistent update set affects a location in the shared name space, which is distinct from the locations affected by the other updates.

Parallel evaluation of guards

An evolving algebra program can contain any finite number of rules. Each of these rules is guarded by a condition.

if c_1
then U_1
.
.
.
if c_n
then U_n

To determine which of the available rules are applicable in a given state, the conditions $c_1, .., c_n$ must be evaluated. These evaluations can take place in parallel, as child processes of the process which determines the applicable rules in a program.

So, in the core theory of evolving algebras, two general kinds of parallelism occurs: parallel execution of updates, and parallel evaluation of expressions. The latter kind occurs in three situations: evaluation of complex terms or conditions, evaluation of left and right hand sides of local function updates, and evaluation of guards to determine sets of applicable transition rules.

3.9.3 Modules and parallelism

In the core theory, conditions, terms and updates are of limited complexity. More specifically, they are always defined in terms of other conditions, terms and updates, never in terms of complete evolving algebras. As a consequence, parallel evaluation of expressions and parallel execution of updates involve child processes of limited size.

In the modularized theory of evolving algebras, the limit on the complexity of conditions, terms and updates is lifted. They can be defined in terms of functional and procedural modules, which both encapsulate complete evolving algebras. Consequently, the child processes involved in parallel evaluation of expressions and parallel execution of updates are computations of full evolving algebra runs.

Modules and parallel update execution

Procedural modules are abstractions over updates. Correspondingly, procedural module invocations are updates. Suppose the following updates occur in an update set:

$$(f, a) := proc$$
$$f(y) := z$$

When this update set is executed, the evolving algebra encapsulated in *proc* will run in parallel with the execution of the local function update.

In an update set, more than one procedural update invocation may occur.

$$(f, a) := proc$$
$$(g, b) := proc$$

When these updates are executed, two instantiation of the evolving algebras encapsulated by *proc* will run in parallel.

The consistency update imposed on update sets will ensure that evolving algebras running in parallel do not interfere with each other.

Modules and parallel expression evaluation

Functional modules are abstractions over terms. Hence, functional module invocations are terms. Assume *func* is the name of a functional module, and consider the following complex term:

$$f(a, func(b))$$

The evaluation of this complex term involves the parallel evaluation of its subterms. Therefore, the evolving algebra encapsulated by *func* will run in parallel with the evaluation of a.

Several subterms of a complex term can be functional module invocations. For example:

$$f(func(a), func(b))$$

When this complex term is evaluated, two instantiations of the evolving algebra encapsulated by *func* will run in parallel.

Similar situations occur when functional module invocations appear in transition rule guards or in left and right hand sides of local function updates. The purely functional nature of functional modules insures that evolving algebras running in parallel will never interfere with each other.

3.9.4 Parallel and distributed evolving algebras in the literature

Massive parallelism

Gurevich [Gur95] has shown how the introduction of variable declarations into evolving algebra programs makes the specification of *massive* parallel computational processes possible. For instance, an update[3] of the following form:

var v **ranges over** s
with U

[3] Gurevich does not strictly separate updates from rules. Consequently, his variable declarations are rules with nested rule sets, in stead of updates with nested update sets.

specifies that the updates in update set U, in which the variable v may occur free, be executed for every element in sort s. Hence, variable declarations serve as universal quantors over the elements of a sort. If the sort s has an infinite number of elements, the variable declaration accomplishes *unbounded* parallelism. However, it is advisable to restrict s to dynamic sorts, which have finite numbers of elements.

Note that a variable declaration can be used to form consistent update sets only if the union of the instantiation of U for every element in s is a consistent update set. Consider the following variable declaration:

var v **ranges over** s
with $a := v$

This variable declaration leads to inconsistency, unless s contains no more than one element.

Gurevich' massive parallelism is compatible with the modularization of evolving algebras. When the update set U contains functional or procedural modules, massive number of evolving algebras will run in parallel.

Distributed evolving algebras

synchronization and communication protocols are explicitly needed This section has been concerned with parallel processes, not with distributed processes. Several proposals have been made to introduce distributedness into the evolving algebra theory. The most comprehensive proposal is given by Gurevich in [Gur95].

Gurevich introduces distributed evolving algebras by having not just a single program, but a finite collection of programs. If we imagine a program being executed by an agent, this generalization amounts to the introduction of multiple agents in stead of a single agent. The vocabularies of the various programs are partially overlapping. The agents identify themselves using a special zero-argument function called *self*. To allow cooperative actions, the agents are organized into *teams*, which are agents themselves. Multi-agent evolving algebras have been used to model communication protocols for distributed systems [Gur95].

3.10 Modules and structured evolving algebra design

The introduction of modules into the theory of evolving algebras makes adequate treatment of large evolving algebras possible. Modules allow computations to be subdivided into relatively independent subcomputations. These subcomputations can be specified in isolation in separate modules. Using such modules as building blocks, well-structured evolving algebras can be formulated.

Large evolving algebras with modular structures are easier to formulate, understand and adapt. Furthermore, testing of large evolving algebras and reasoning about them is facilitated by a modular structure. In particular, the modularization of evolving algebras makes it possible to adopt a structural design methodology for evolving algebras, and to construct modular proofs about evolving algebras.

3.10.1 Modular evolving algebra design

A top-down design methodology can be applied to construct modular evolving algebras. According to this methodology, one starts by formulating a top-level module which reflects the global functionality. Subsequently, detail can be added to the specification by constructing the specifications of the modules that are invoked by the top-level module. This process can be repeated until the lowest level of modules has been reached.

Modularization enables re-use of parts of evolving algebra specifications. The same module can be used in different evolving algebras.

Modularization makes modifying evolving algebra specifications easier. Specifications need not be modified as a whole, but changes can be made locally inside modules. Changes inside modules are hidden within the module. Hence, no other parts of the evolving algebra need to be changed.

3.10.2 Modular evolving algebra proofs

The modular structure of an evolving algebra can be exploited when proofs about them are constructed. For instance, the proof that a particular modular evolving algebra terminates, can be constructed from the termination proofs of its constituent modules. In general, proofs about modules can be utilized as premisses for proofs about evolving algebras that invoke them.

When modules are re-used, the proofs that have been constructed about them can be re-used as well. Local modifications of evolving algebras result in local modifications of proofs.

Correctness proofs for evolving algebra modules take different forms for functional and procedural modules. To prove a functional module correct, one must prove that it implements the (mathematical) function it is required to implement. To prove a procedural module correct, one must prove that it implements a certain state transformation. It must be proven that, assuming a given pre-condition obtains before the procedural module invocation takes effect, a certain post-condition will be satisfied after the module has been executed.

Inductively confirming hypotheses about evolving algebras, i.e. testing, benefits from modularization as well as deductive proof construction. Each module can be tested and corrected in isolation before the evolving algebra is tested in its entirety.

In this chapter, an extension of the core theory of evolving algebras with functional and procedural modules has been formulated. Also, the benefits of modularization with respect to structured evolving algebra design and parallelism have been described. In the upcoming chapter, the use of modules will be demonstrated by example.

Chapter 4

A modular evolving algebra for lambda reduction

4.1 Lambda reduction 59
4.2 Graph representation of lambda expressions 61
4.3 Modular structure of the evolving algebra 62
4.4 Top level module 62
4.5 Module for finding the redex 64
4.6 Module for reducing the redex 66
4.7 Module for constructing the reduct 68

In the previous chapter, the theory of evolving algebras was enhanced with modules. In this chapter a modular evolving algebra which models graph reduction of lambda expressions, is presented to illustrate the use of modules. In the first section, a brief survey of lambda reduction in general will be given. In the second section, the graph representation of lambda expressions used by our evolving algebra, is explained. In the sections 4.3 to 4.7 the evolving algebra itself is piecewise specified and discussed.

4.1 Lambda reduction

In this section a condensed account of lambda reduction is given to serve as a basis for the subsequent sections. A more detailed account of the lambda calculus can be found in Barendregt [Bar84].

Lambda reduction is the process of rewriting lambda expressions to simpler ones. We will only consider expressions in the *pure* lambda calculus, which have the following syntax:

$$\langle exp\rangle \rightarrow \langle var\rangle | \lambda\langle var\rangle.\langle exp\rangle | \langle exp\rangle\langle exp\rangle | (\langle exp\rangle)$$
$$\langle var\rangle \rightarrow x|y|z|\ldots$$

For example, the following are lambda expressions:

$$\lambda x.\lambda y.xyy$$
$$(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))$$

A lambda expression of the following form:

$$(\lambda V.B)A$$

where B and A are arbitrary lambda expressions and V is a variable, is called a reducible expression, or *redex*. B is called the body and A is called the argument of the redex.

Any lambda expression which contains a redex as a subexpression can be reduced by replacing this redex by its body B in which all occurrences of its variable V have been replaced by its argument A[1]. This prescription for reduction is called the β-rule. It can be schematically rendered as follows:

$$\ldots(\lambda V.B)A\ldots \rightarrow_\beta \ldots B[A/V]\ldots$$

For instance, we have the following β-reductions:

$$(\lambda x.\lambda y.y)\underline{((\lambda z.zz)(\lambda z.zz))} \rightarrow_\beta (\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))$$
$$\underline{(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))} \rightarrow_\beta \lambda y.y$$

The underlining in the expressions on the left hand side indicate which redex is being reduced.

When reductions are carried out in succession, reduction sequences are constructed. A reduction sequence ends when an expression is produced which contains no more redexes. This last expression is said to be in *normal form*. Alternatively, one may decide to reduce only until an expression is produced which is in *weak head-normal form*. This is the case if the expression satisfies the following conditions:

- the expression is a variable, or
- the expression is of the form $\lambda V.E$, or
- the expression is of the form $xE_1E_2\ldots E_n$.

If one pictures lambda expressions as trees, one can concisely say that an expression is in weak head-normal form if its left spine does not contain any redexes. Some lambda expressions can be reduced to an expression in weak head-normal form, but not to one in ordinary normal form. Some expressions can be reduced to neither. In this latter case, the reduction sequence is non-terminating. In this paper we will limit ourselves to reductions to weak head-normal form. For sake of brevity we will sometimes drop the qualification "weak head", but ordinary normal form is never intended.

Since a lambda expression may contain more than one redex, the reduction steps may involve choosing between possible redexes. These choices can be made according to a particular reduction *strategy*. Two strategies are of particular interest:

Applicative-order reduction At each step, the *leftmost innermost* redex is chosen. A redex is called innermost if none of its subexpressions are redexes. A redex is called leftmost if it occurs to the left of all other redexes.

[1] We will assume all variables to be uniquely chosen, to rule out the possibility of name clashes.

Normal-order reduction At each step, the *leftmost outermost* redex is chosen. A redex is called outermost if it is not a subexpression of some other redex. The qualification "outermost" is actually redundant. All leftmost outermost redexes are leftmost redexes and *vice versa*. Consequently, normal-order reduction is sometimes called leftmost reduction [Bar84, 180].

Whether the reduction sequence that is constructed terminates or not may depend on which reduction strategy is adopted. The set of lambda expressions whose reduction terminates is more extensive for normal-order reduction than for applicative-order reduction. In fact, normal-order reduction is *normalizing*, i.e. it always produces a normal form if there is one[2], but applicative-order reduction is not.

In this section, a brief survey of lambda reduction in general was given. The evolving algebra to be presented in sections 4.3 to 4.7 will be limited to normal-order reduction to weak head-normal form. Also, the evolving algebra will presuppose a parser and operate on graph representations of lambda expressions. The next section explains the graph representation to be used.

4.2 Graph representation of lambda expressions

In the evolving algebra to be presented in the next section, lambda expressions will be represented by binary graphs. The nodes of these graphs are elements of a dynamic sort *Node*. To these nodes, four dynamic functions are applicable: *nodetype*, *left*, *right*, and *name*. For each node n the value of *nodetype*(n) is either *lambda*, *apply* or *variable*, indicating whether it represents a lambda expression, an apply expression or a variable. If the value of *nodetype*(n) is *lambda* or *apply*, then the values of $left(n)$ and $right(n)$ are the nodes which represent the left and right subexpressions of the expression it represents. If the value of *nodetype*(n) is *variable*, then $left(n)$, and $right(n)$ are undefined, and the value of *name*(n) is the name of the variable which is represented by n.

This graph representation of lambda expressions conforms to the syntax diagram of section 4.1, except for the parentheses. These are not present in the graph representation, because no ambiguity of scope can occur.

The graph representation will differ from the syntax diagram on a second account because we will allow *sharing*. A node is said to be shared if it is a child of more than one parent. When a node is shared the subexpression it represents has several occurrences which are identical to each other.

The evolving algebra for lambda reduction will need to traverse the graphs that represent lambda expressions. Traversal involves on the one hand *descending* the graph, by switching from a node to one of its children, and on the other hand *backtracking*, which consists in switching from a node to its parent. To make backtracking possible, a fifth dynamic function on nodes is needed, called *parent*. At each descending step from a node n to its child node m, the value of $parent(m)$ will be set to n. This value is again retrieved to backtrack from m in a later stage. For the root node of a graph, the value of *parent* is always undefined.

In the next section, the evolving algebra for graph reduction of lambda expressions will be presented, which makes use of the graph representation explained in this section.

[2] A proof that normal-order reduction is normalizing can be found in Barendregt [Bar84, 326ff].

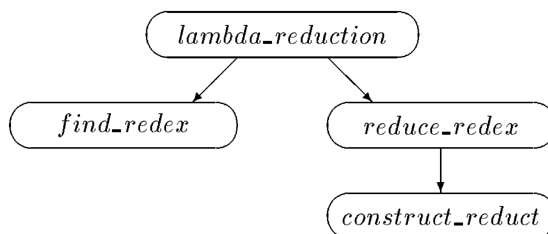


Figure 4.1: Modular structure.

4.3 Modular structure of the evolving algebra

In this section, a modular evolving algebra for normal-order graph reduction of lambda expressions to weak head-normal form will be given. In [FH88] an informal description of such an algorithm can be found.

In outline, the algorithm operates as follows. Starting from an initial lambda expression, a reduction sequence is produced by performing single β-reductions in succession, until an expression in weak head-normal form is obtained. Each β-reduction consists in finding the leftmost outermost redex and replacing this redex by its body, in which all occurrences of its variable have been replaced by its argument.

We have designed a modular evolving algebra for this algorithm, which consists of four modules. These modules are arranged in a hierarchy. The top level module is called *lambda_reduction*. The immediate subordinates of this top level module are the modules *find_redex* and *reduce_redex*. These modules perform the subtasks of finding a redex and reducing a redex to its reduct respectively. The top level module *lambda_reduction* invokes these modules in alternation, until no more redexes can be found. The second subordinate module, *reduce_redex* delegates the task of constructing a redex to a further subordinate called *construct_reduct*. After invoking this module, the module *reduce_redex* replaces the redex by its reduct. This module structure is pictured in figure 4.1.

In the following sections, the four modules of our evolving algebra will be specified and discussed. We will adopt a top down order of presentation.

4.4 Top level module

The top level module is a procedural module, as are all modules in our evolving algebra. Through its parameters it receives a graph representation of a lambda expression. The root of this graph is pointed at by the 0-ary static function parameter *expr*. Thus, the header of the top level module is as follows:

module lambda_reduction (**ds** $Node$
df $expr : Node$
$type : Node \rightarrow NodeType$
$name : Node \rightarrow Variable$
$left : Node \rightarrow Node$
$right : Node \rightarrow Node$)

The module operates by alternating between two modes, called *find_redex* and *reduce_redex*. In the first mode, the redex of an expression is determined by calling

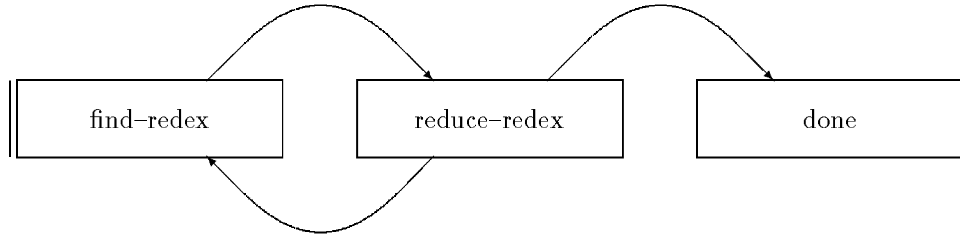


Figure 4.2: Mode structure of the module *lambda_reduction*.

the subordinate module *find_redex*. (We have given the module and the mode in which it is invoked the same name.) In the second mode, the redex which has been found is reduced by invoking the module with the name *reduce_redex*. If no more redexes can be found, a third mode is entered, called *done*. The mode structure of the top level module is depicted in figure 4.2. These three modes are elements of the static sort *Mode*. The dynamic function *mode* is used to keep track of the current mode.

static sorts
$Mode \;\; \Rrightarrow \;\; (find_redex, reduce_redex, done)$
dynamic functions
$mode : Mode$

In the start update set, the dynamic function *mode* is initialize as *find_redex*.

Start:
$mode := find_redex$

In the final state of the evolving algebra, the value of *mode* should be *done*. This is expressed by the stop condition:

Stop: $mode = done$

We will specify the guards of the rules of the module in such a way that the stop condition is guaranteed to be satisfied when no more transition rules are applicable.

The program of the module contains three rules. The names of these rules are indicative of the mode in which they are applicable:

Program:
find_redex
if $mode = find_redex$
then $(Node; expr, redex, found, left, type) := find_redex$
$mode := reduce_redex$
reduce_redex_a
if $mode = reduce_redex \wedge found$
then $(Node; redex, type, left, right, name) := reduce_redex$
$mode := find_redex$
reduce_redex_b
if $mode = reduce_redex \wedge \neg found$
then $mode := done$

The first rule is applicable in the mode $find_redex$. When this rule is fired, the module $find_redex$ is invoked. This module updates two 0-ary functions of the following types:

dynamic functions
$redex : Node$
$found : Boolean$

The dynamic function $redex$ is updated by the module $find_redex$ to point to the root of the leftmost outermost redex of the expression pointed at by $expr$. If no redex can be found, $found$ will be updated to $false$, otherwise to $true$. In parallel, the mode will be set to $reduce_redex$.

In the mode $reduce_redex$ either one of two rules can be applicable. The first rule is applicable if a redex has been found. The update set of this rule invokes the module $reduce_reduct$ to reduce the redex. This subordinate module operates on the subtree pointed at by $redex$. In parallel the mode is set to $find_redex$ again, to continue the find-reduce cycle in the next transition.

If no redex has been found in the mode $find_redex$, the third rule will fire in mode $reduce_redex$. Its effect is to end the find-reduce cycle by updating $mode$ to $done$. In the mode $done$, no transition rules are applicable, and the top level module terminates.

For completeness, we present the complete signature of body of the top level module:

Signature:
static sorts
$Mode \Rrightarrow (find_redex, reduce_redex, done)$
$NodeType \Rrightarrow (lambda, apply, variable)$
dynamic functions
$redex : Node$
$found : Boolean$
$mode : Mode$

We will now turn to the subordinate modules.

4.5 Module for finding the redex

The task of the subordinate module $find_redex$ is to find the leftmost outermost redex of a lambda expression. It receives the graph representation of the lambda expression to be searched through its parameters $Node$, $left$, and $type$. The dynamic function parameters $redex$ is to be updated by the module to the root of the redex that is found. The dynamic function parameter $found$ is used to flag whether the search for a redex was successful. Thus, the header of the module is as follows:

module find_redex (**ds** $Node$
df $expr : Node$
$redex : Node$
$found : Boolean$
$left : Node \rightarrow Node$
$type : Node \rightarrow NodeType$)

If no redex is found by the module, then the expression is in weak head-normal form.

As in the case of the top-level module, the operation of the module $find_redex$ is guided by modes. Only two modes are distinguished: not_done and $done$. The signature therefore contains the following declarations:

static sorts
$Mode \;\Rrightarrow\; (not_done, done)$
dynamic functions
$mode : Mode$

Note that these names are local. They should not be confused with the same names used in the top level module and the other modules. In the start update set, the mode is updated to not_done, and in the final states, $mode$ should equal $done$.

Stop: $mode = done$

The guards of the transitions rule to be presented shortly, guarantee the stop condition to be true in all final states.

The module $find_redex$ operates by descending the spine of the graph pointed at by $expr$ in a step-wise fashion. When a redex is found along this spine, the mode $done$ is entered, $redex$ is updated to point to this redex, and the flag $found$ is set to $true$. If the end of the spine is reached without finding a redex, then the mode $done$ is entered and the flag $found$ is set to $false$. For the step-wise descent along the spine, a pointer called $current$ is used.

dynamic functions
$current : Node$

In the start update set, this pointer is set to the root of the graph: $expr$. Hence, the update set of the module performs two updates:

Start:
$current := expr$
$mode := not_done$

The program of the module $find_redex$ contains four rules:

Program:

if $mode = not_done \wedge type(current) = apply \wedge type(left(current)) = lambda$
then $redex := current$
$found := true$
$mode := done$

if $mode = not_done \wedge type(current) = apply \wedge type(left(current)) = apply$
then $current := left(current)$

if $mode = not_done \wedge type(current) = apply \wedge type(left(current)) = variable$
then $mode := done$
$found := false$

if $mode = not_done \wedge type(current) \neq apply$
then $mode := done$
$found := false$

We have not given these rules names. All these rules are applicable only if the mode is *not_done*. The first rule fires when the current node pointed at by *current* is a redex. Such a redex is recognized by the fact that its root is an apply expression, which has a lambda expression as its first argument.

The second rule is applicable if no redex is pointed at by *current*, while the end of the spine has not yet been reached. The update set of this rule moves the pointer *current* one step further down the spine.

The last two rule are applicable, when the end of the spine is reached without finding a redex. In this case the expression is already in weak head-normal form. The update sets of these rules set the flag *found* to *false*.

The complete signature of this module is as follows:

Signature:

static sorts

$Mode \Rrightarrow (not_done, done)$

$NodeType \Rrightarrow (lambda, apply, variable)$

dynamic sorts

$Node$

static functions

$lambda : NodeType \Rrightarrow lambda$

$apply : NodeType \Rrightarrow apply$

$variable : NodeType \Rrightarrow variable$

dynamic functions

$mode : Mode$

$current : Node$

We will now turn two the next subordinate module of *lambda_reduction*, which is invoked when a redex has been found by *find_redex*.

4.6 Module for reducing the redex

The task of the module *reduce_redex* is to replace the redex of an expression by its reduct. This task is performed in two steps. First, the subordinate module *construct_reduct* is invoked to construct the reduct for the given redex. Second, the redex is 'overwritten' by the reduct, i.e. the node pointed at by *redex* is made the root of the graph which represents the reduct.

The redex pointer is transferred to the module *reduce_redex* through a static function parameter. The graph representation of the redex pointed at is received through the dynamic parameters *Node*, *type*, *left*, *right* and *name*. The header is as follows:

module reduce_redex (**ds** $Node$
df $redex : Node$
$type : Node \rightarrow NodeType$
$left : Node \rightarrow Node$
$right : Node \rightarrow Node$
$name : Node \rightarrow Variable$)

The module *reduce_redex* operates in three modes: *construct_reduct*, *overwrite_redex*, and *done*.

static sorts
$Mode \Rrightarrow (construct_reduct, overwrite_redex, done)$
dynamic functions
$mode : Mode$

These modes are entered consecutively. There is no iteration within a mode, nor alternation between modes. The module starts in mode *construct_reduct* and terminates in mode *done*.

Start:
$mode := construct_reduct$

Stop: $mode = done$

The program consists of two rules:

Program:

construct_reduct
if $mode = construct_reduct$
then $(Node; redex, type, left, right, name, reduct) := construct_reduct$
$mode := overwrite_redex$

overwrite_redex
if $mode = overwrite_redex$
then $type(redex) := type(reduct)$
$left(redex) := left(reduct)$
$right(redex) := right(reduct)$
$name(redex) := name(reduct)$
rem $reduct : Node$
$mode := done$

In the first mode, the first rule is applicable. This rule invokes *construct_reduct*, and switches to mode *overwrite_redex*. As a result, the pointer *reduct* will point at the root of the reduct of *redex*. The second rule is applicable in the second mode. The update set of this rule redirects all pointers emanating from the redex root-node to the values of the corresponding pointers of the reduct root-node. In parallel, the reduct root-node, which has become superfluous, is removed.

In full, the signature of the module *reduce_redex* is as follows:

Signature:

static sorts
$Mode \Rrightarrow (construct_reduct, overwrite_redex, done)$
$NodeType \Rrightarrow (lambda, apply, variable)$
dynamic sorts
$Node$
static functions
$lambda : NodeType \Rrightarrow lambda$
$apply : NodeType \Rrightarrow apply$
$variable : NodeType \Rrightarrow variable$
dynamic functions
$mode : Mode$
$reduct : Node$

In the next section we will present the last of the four modules that constitute the modular evolving algebra for lambda reduction.

4.7 Module for constructing the reduct

The task of the module *construct_reduct* is to construct the graph representing the reduct of a given redex. The graph structures are transferred into and outof the module through the dynamic parameters *Node*, *type*, *left*, *right*, and *name*. The pointers *redex* and *reduct* are used to point to the root node of the redex and the reduct, respectively. The header of *construct_reduct* is as follows:

module construct_reduct (**ds** $Node$
df $redex : Node$
$type : Node \rightarrow NodeType$
$left : Node \rightarrow Node$
$right : Node \rightarrow Node$
$name : Node \rightarrow Variable$
$reduct : Node$)

The mode structure of *construct_reduct* is similar to the mode structure of *find_redex*. Two states are distinguished: *not_done* and *done*.

static sorts
$Mode \Rrightarrow (not_done, done)$
dynamic functions
$mode : Mode$

In the first mode, the graph which represents the body of the redex is traversed in depth-first fashion. During this traversal, the body of the redex is copied. At the same time, the pointers in the copy to variables bound by the lambda of the redex are redirected to the argument of the redex. When the traversal of the tree is completed, the mode *done* is entered.

During the traversal, three auxiliary pointers are used:

dynamic functions
$reduct : Node$
$body : Node$
$parent : Node \rightarrow Node$

The pointers *reduct* and *body* are used to point at the current node in the reduct graph and the current node in the graph representing the body of the redex, respectively. These pointers move through their respective graphs in a synchronized fashion, i.e. they always move simultaneously in the same direction. The *parent* pointer is used to enable backtacking. At each move down the graph, a *parent* pointer is installed from the child node that becomes current, to the parent node which ceases to be current. When backtracking is required, the *parent* pointer is inspected to regain the parent node of the current node.

In the start update set, several updates are performed to initialize the module.

Start:
$body := right(left(redex))$
$arg := right(redex)$
$var := name(left(left(redex)))$
new $n : Node$ **with**
$reduct := n$
$type(n) := type(right(left(redex)))$
$mode := not_done$

The first three of these updates create temporary pointers to the body, the argument and the binding variable of the redex, respectively. The extend update creates the root node of the reduct graph, and initializes its type. The last update initializes the mode as *not_done*, to start the copying of the body.

During the traversal of the graphs, several situations can occur. The program contains a rule to handle each situation. We will discuss these rules in groups. The three rule in the first are applicable in situations where an occurence of the variable bound by the redex is encountered. In these situations the following condition is satisfied:

$$type(body) = variable \wedge name(body) = var$$

In each of these three situations, the argument *arg* of the redex must be substituted somehow for the variable. The situations differ in the relation of the current reduct-node to its parent. In the first situation, the current-reduct node is the left child of its parent, in the second situation it is the right child, and in the third situation, the current node does not have a parent at all. The rules fore these situations are:

if $mode = not_done \wedge type(body) = variable \wedge name(body) = var \wedge$
$defined(parent(reduct)) \wedge left(parent(reduct)) = reduct$
then $left(parent(reduct)) := arg$
$reduct := parent(reduct)$
$body := parent(body)$

if $mode = not_done \wedge type(body) = variable \wedge name(body) = var \wedge$
$defined(parent(reduct)) \wedge right(parent(reduct)) = reduct$
then $right(parent(reduct)) := arg$
$reduct := parent(reduct)$
$body := parent(body)$

if $mode = not_done \wedge type(body) = variable \wedge name(body) = var \wedge$
$\neg defined(parent(reduct))$
then $reduct := arg$
$mode := done$

In the first two situations, backtracking occurs, and the argument becomes the left or right child of the new current reduct node. In the third situation, the argument becomes the new reduct, and the traversal is complete.

In the second group of possible situations, a variable is encountered which is different from the variable bound by the redex. In other words, the following condition is satisfied:

$$type(body) = variable \wedge name(body) \neq var$$

In this case, no substitution needs to be performed. Depending on whether the current node has a parent, one of the following two rules is applicable:

> **if** $mode = not_done \wedge type(body) = variable \wedge name(body) \neq var \wedge$
> $\quad defined(parent(reduct))$
> **then** $name(reduct) := name(body)$
> $\quad reduct := parent(reduct)$
> $\quad body := parent(body)$

> **if** $mode = not_done \wedge type(body) = variable \wedge name(body) \neq var \wedge$
> $\quad \neg defined(parent(reduct))$
> **then** $name(reduct) := name(body)$
> $\quad mode := done$

The update set of the first of these rules copies the variable from the body to the reduct, and backtracks. The update set of the second rule performs the same copy operation, but switches to mode *done* in stead of backtracking.

In the last group of four possible situations, a node is encountered of a different type than variable. Thus, the following condition is satisfied:

$$type(body) \neq variable$$

If the left child of the current node has not been visited yet, the following rule applies:

> **if** $mode = not_done \wedge type(body) \neq variable \wedge$
> $\quad \neg defined(left(reduct))$
> **then** $body := left(body)$
> $\quad parent(left(body)) := body$
> $\quad$ **new** $n : N$ **with**
> $\quad\quad left(reduct) := n$
> $\quad\quad reduct := n$
> $\quad\quad parent(n) := reduct$
> $\quad\quad type(n) := type(left(body))$

The update set of this rule creates a new node in the reduct-graph. This new node becomes the left child of the current node. Simultaneously, the reduct pointer and the body pointer descend to their left children, and parent pointers are installed to enable backtracking in a later stage. Also, the type of the left child of the current body node is copied into the new reduct node.

If the left child of the current node has been visited, but the right node has not, the following rule is triggered:

> **if** $mode = not_done \wedge type(body) \neq variable \wedge$
> $\quad defined(left(reduct)) \wedge \neg defined(right(reduct))$
> **then** $body := right(body)$
> $\quad parent(right(body)) := body$
> $\quad$ **new** $n : Node$ **with**
> $\quad\quad right(reduct) := n$
> $\quad\quad reduct := n$
> $\quad\quad parent(n) := reduct$
> $\quad\quad type(n) := type(right(body))$

The update set of this rule operates on the right children of the reduct and body graph in the same way as the previous rule operated on the left children.

If both children of the current node have been visited, one of the following rules is applicable:

if $mode = not_done \wedge type(body) \neq variable \wedge defined(left(reduct)) \wedge$
$defined(right(reduct)) \wedge defined(parent(reduct))$
then $reduct := parent(reduct)$
$body := parent(body)$

if $mode = not_done \wedge type(body) \neq variable \wedge defined(left(reduct)) \wedge$
$defined(right(reduct)) \wedge \neg defined(parent(reduct))$
then $mode := done$

The first of these rule is fired when the current node has a parent. Its update set backtracks. The second rule triggers when no parent exists, and the traversal is complete. Its update set switches to the mode *done*.

This completes the program of the module *construct_reduct*. The signature of this module is as follows:

Signature:

static sorts
$Mode \Rrightarrow (not_done, done)$
$NodeType \Rrightarrow (lambda, apply, variable)$
static functions
$lambda : NodeType \Rrightarrow lambda$
$apply : NodeType \Rrightarrow apply$
$variable : NodeType \Rrightarrow variable$
dynamic functions
$body : Node$
$arg : Node$
$var : Variable$
$mode : Mode$
$parent : Node \rightarrow Node$

Now all four modules of our modular evolving algebra for lambda reduction have been presented, we will briefly review its operation. The top level module *lambda_reduction* performs a find-reduce cycle until the graph which represents a lambda expression has been brought into weak head-normal form. In the first step of the find-reduce cycle, the subordinate module *find_redex* is invoked. This module descends the left spine of the graph searching for a redex. The first redex encountered is the leftmost-outermost redex of the expression. In the second step of the find-reduce cycle, the module *reduce_redex* is invoked. This module in its turn invokes the module *construct_reduct* to construct the reduct of the redex that was found. Then, the module *reduce_redex* overwrites the root node of the redex with the root node of its redex. In this fashion, one reduction step is completed by each find-reduce cycle. When no redex is found in the first step, reductions sequence ends with an expression in weak head-normal form, and the modular evolving algebra is finished.

Part II

Automated support for evolving algebras

The formulation of specific evolving algebras is not an algorithmic task, but a creative process. Still, many subtasks of the evolving algebra designer could be supported by appropriate algorithmic tools. The development of such tools is one of the aspects of the evolving algebra programme.

The second part of this report is dedicated to automated support of evolving algebras. In particular, this part presents the evolving algebra tool EVADE. Two groups of three chapters can be distinguished within this part. The first three chapters deal with EVADE and other evolving algebra tools from the standpoint of a user. The second set of three chapters jointly describe the implementation of EVADE.

Chapter 5

Inventory of evolving algebra support

5.1 Evolving algebra compilers 76
5.2 Evolving algebra run analyzers 77
5.3 Interactive theorem provers 77
5.4 Evolving algebra transformers 78

In this chapter we will investigate the possibilities and actualities of automated evolving algebra support. We will make a categorization of conceivable evolving algebra tools, which comprises the following categories:

- Evolving algebra compilers or interpreters.
- Evolving algebra run analyzers.
- Interactive theorem provers for evolving algebras.
- Evolving algebra transformers.

This categorization is not presumed to be exhaustive. In the following sections, each category will be briefly characterized, and existing tools that fall into the categories will be listed.

5.1 Evolving algebra compilers

The language of evolving algebras can be used as a programming language as well as a specification language (see section 1.3.2). Hence, in order to be able to run evolving algebras on concrete machines, a compiler (or interpreter) is needed. The task of evolving algebra compilers is to transform an evolving algebra specification to executable code.

Two evolving algebra interpreters have been developed.

- The DASL-ALMA compiler. In [Kap93] Kappel defines an evolving algebra specification language, called DASL, and designs an abstract machine, called ALMA, which is a Prolog program. A compiler has been implemented in Prolog that compiles DASL-specifications to ALMA-instructions. Hence, the combination of this DASL-to-ALMA compiler and the ALMA machine forms a DASL-to-Prolog compiler. In the sequel, we will in short refer to this compiler as the DASL-compiler.

- leanEA. In [BP95] Beckert and Posegga describe an extremely small Prolog program which turns the Prolog interpreter itself into a virtual machine on which evolving algebras can be run. Hence, the Prolog interpreter extended with this Prolog program constitutes an evolving algebra interpreter. This interpreter is called leanEA. In the same paper, a second version of lean$EA$is presented, which accepts modularized evolving algebras.

5.2 Evolving algebra run analyzers

To provide feedback to evolving algebra specification, it is desirable to be able to investigate the behavior of an evolving algebra during its run. Hence, a run analyzer is needed, which allows one to run an evolving algebra step by step, and to inspect its intermediate states.

Two evolving algebra run analyzers have been realized.

- In [HM] Huggins and Mani describe an evolving algebra run analyzer which is implemented and embedded in the programming language C. We will refer to this tool as the C run analyzer.

- In [Die95b] Diesen describes an evolving algebra run analyzer which is implemented and embedded in the programming language Scheme. We will call it the Scheme run analyzer.

5.3 Interactive theorem provers

An evolving algebra run analyzer is useful for testing, investigating and debugging evolving algebras. However, one does not only want to inductively confirm hypotheses about evolving algebras, but even deductively prove them. For this purpose, an interactive theorem prover can be helpful. The task of an interactive theorem prover is to assist its user in constructing proofs for theorems about specific evolving algebras. This assistance should involve automatic generation of subproofs, application of proof steps indicated by the user, and organized presentation of constructed (partial) proofs. Prerequisite to the development of a theorem prover is the formulation of a proof theory for evolving algebras. For simplified one-sorted evolving algebras an attempt at formulating such a proof theory has been made by Poetzsch-Heffner in [PH94]. But for general evolving algebras no proof theory is currently available, and no interactive theorem provers have been built.

Somewhat less ambitiously, theorem provers could be developed that are dedicated to specific theorems. For instance, tools could be constructed for proving that a given evolving algebra is consistent, or that it is deterministic, or that it terminates for arbitrary input. At the moment, no such dedicated provers exist.

5.4 Evolving algebra transformers

In [Die95a] Diehl defines a number of transformations on evolving algebras. Among these transformations are macro expansion and transition rule flattening. All these transformations are optimizing techniques with which a given evolving algebra can be transformed to an operationally equivalent evolving algebra that is more efficient.

Another example of a transformation on evolving algebras is partial evaluation. A partial evaluator specializes the specification of an evolving algebra for specific input, thus generating a specification of a specialized evolving algebra. Partial evaluation is especially useful in case of evolving algebras which specify programming language compilers. Specialization of such an evolving algebra for a specific source program yields an evolving algebra specification for this program. Partial evaluation has been defined by Huggins in [Hug95]. A partial evaluator has been implemented in C.

We have listed four kinds of evolving algebra tools: compilers, run analyzers, theorem provers, and transformers. Implementations of some of these kinds of tools have already been realized. Unfortunately, all these implementations have some shortcomings. For instance, the DASL-compiler does not accept modular evolving algebras and does not provide the evolving algebras it accepts with an interface to their environment. Neither version of lean*EA* supports dynamic sorts. The run analyzers implemented in C and Scheme do not offer any support for non-deterministic evolving algebras. The Scheme run analyzer requires evolving algebra to be specified in a syntax that deviates considerably from the conventional evolving algebra notation. The C run analyzer requires static functions to be specified in a way that is inelegant and quite deviant from mathematical usage.

A further shortcoming of the tools implemented so far is that they all require evolving algebras to be specified in different notations, and that they operate in isolation from each other. It is desirable for an evolving algebra designer to be able, for example, to compile an evolving algebra whose run he has just analyzed, and to subsequently use the compiled evolving algebra as a module inside a larger evolving algebra. Therefore, evolving algebra tools from different categories would preferably be integrated into a single evolving algebra development environment.

In short, there is a demand for an evolving algebra development environment, that satisfies the following minimum requirements:

- Modularized evolving algebras are accepted.
- Many-sorted evolving algebras are accepted.
- Dynamic sorts are allowed.
- Non-determinism is supported.
- Static functions and sorts can be specified elegantly.
- A compiler and a run analyzer are available that can be used in combination.

In the next section we will present EVADE, an evolving algebra development environment that meets these minimum requirements.

Chapter 6

EvADE: an evolving algebra tool

6.1 Overview 79
6.2 Example 81
6.2.1 Preparing EvADE and the evolving algebra 82
6.2.2 Using EvADE on the evolving algebra 83
6.3 Advanced features 86
6.3.1 Static function definitions 86
6.3.2 Static sort definitions 87
6.3.3 Modularized evolving algebras 87
6.3.4 Non-determinism 88
6.4 Summary 90

EvADE is an evolving algebra tool developed by the author. It comprises an evolving algebra compiler and an evolving algebra run-analyzer. The tool is embedded and implemented in the functional programming language Gofer. This chapter will describe EvADE from the viewpoint of a user.

6.1 Overview

The evolving algebra formalism is embedded in standard mathematics. More specifically, evolving algebra specifications make use of mathematical sets and functions to serve as values of sort identifiers and function identifiers. As a consequence, when designing an evolving algebra tool, it is necessary to decide on a formalism for specifying functions and sets. We have decided to use a strongly typed functional programming language for this purpose: Gofer. A concise exposition of Gofer can be found in chapter 8. Mathematical sets are modelled by Gofer types. Mathematical functions are modelled by Gofer functions. Thus, the evolving algebra tool EvADE is embedded in the functional programming language Gofer.

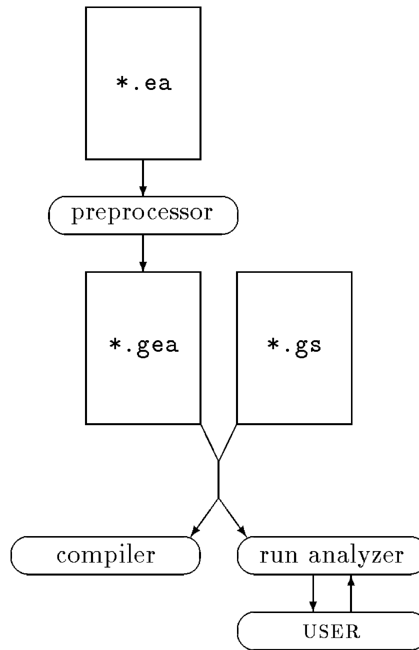


Figure 6.1: Files and components of EVADE.

Input to EVADE is of two kinds: evolving algebra specifications and Gofer definitions. The evolving algebra specifications will be assumed to reside in files with extension `ea`. The Gofer definitions are stored in Gofer script files, which have extension `gs` by convention. The Gofer script files can be used directly by EVADE's compiler and run-analyzer. The evolving algebra specification files, on the other hand, need to undergo preprocessing. The EVADE-preprocessor converts evolving algebra specifications to an intermediary representation, and stores this representation in a file with extension `gea`. This latter file can be read by the compiler and the run-analyzer. The relationships between the files and components of EVADE is depicted in figure 6.1.

The syntax of the evolving algebra specifications accepted by EVADE departs only slightly from the syntax used in part I. We briefly list the main differences:

- Evolving algebras are provided with a functional module header, as described in section 3.5. In these headers, only 0-ary function identifiers can be used as parameters. The types of these identifiers must be described with sort names declared in the signature.
- The keywords `SS`, `DS`, `SF` and `DF` are used in stead of **static sorts**, **dynamic sorts**, **static functions** and **dynamic functions**.
- Instead of the triple arrow $\Rrightarrow$, we use `==>`.
- The arrow $\rightarrow$ is replaced by `->`.
- The cartesian product $A \times B \times C$ is represented by a list notation `(A,B,C)`.
- The logical connectives $\wedge$, $\vee$ and $\neg$ are represented by `/\`, `\/`, and `not`.
- Nested update sets are terminated with a period.

- The remove update is not supported.

The static sort of natural numbers is predefined, as well as a number of functions on natural numbers. They are bound to the Gofer type `Int` and the corresponding Gofer functions on this type.

The absence of remove updates is motivated by a combination of considerations. Remove updates do not perform an essential role in the specification of algorithms. Their utility lies in the management of storage resources. Several possible implementations of the remove update are conceivable. In order to decide between alternative implementations it is necessary to assess the consequences of the various alternatives for future extensions of EVADE. An undeliberated choice of implementation might complicate in particular the extension of EVADE with procedural modules.

6.2 Example

We will now demonstrate the use of EVADE for the example evolving algebra presented in chapter 1. We will assume the following evolving algebra specification to be present in the file `faclist.ea`:

```
MODULE faclist (n : N) head(last):N
SS
DS ListElem
SF
DF i:N
   head : ListElem -> N
   tail : ListElem -> ListElem
   root : ListElem
   last : ListElem
START i := 0
      NEW e : ListElem WITH
        head(e) := 1
        root    := e
        last    := e .
TRANSITION step
     IF i < n
     THEN i := i + 1
          NEW e : ListElem WITH
             head(e)    := head(last) * (i+1)
             tail(last) := e
             last       := e .
STOP i = n
```

Three differences with the evolving algebra specification given in section 1.2 must be noted. Firstly, a functional header has been added. The static function `n` appears in this header instead of in the static function clause. Secondly a stop condition has been added at the bottom. Strictly speaking, this stop condition is redundant; it is satisfied in all possible final states of this particular evolving algebra. For documentation purposes, however, it can be useful. The third difference between the specification given to EVADE and the one presented in section 1.2 is the omission of the names `N`, `+` and `*` from the static sort clause and the static function clause, respectively. This is due to the fact that the natural numbers and operations on them are predefined in EVADE. Note that the keywords `SS` and `SF` can not be omitted even though no static sorts or static functions are declared.

6.2.1 Preparing EVADE and the evolving algebra

We will now walk through the three steps the user of EVADE needs to take in order to load the example evolving algebra into the system.

Step 1: loading EVADE

Once the user has created the above specification and stored it in the appropriate file, an EVADE session can be begun. To begin an EVADE session the user types the following at the Unix prompt:

```
gofer + EvADE.gp
```

(The + indicates that the file to be loaded is a Gofer project file. The conventional extension for these files is gp) As a result, the Gofer interpreter is entered and the EVADE program is loaded into the Gofer interpreter environment. Two functions are now available to the user: `compile` and `analyze`. Shortly, we will explain how these functions are applied to evolving algebras. After this step of loading EVADE, the user finds himself at the Gofer prompt, which is a single question mark.

Step 2: preprocessing an evolving algebra

Before an evolving algebra can be loaded into EVADEand used, it must be preprocessed. To apply the preprocessor to the evolving algebra specification in the file `faclist.ea`, the user must type the following at the Gofer prompt:

```
? preprocess "faclist"
```

The preprocessor will now convert the evolving algebra specification in the file `faclist.ea` to an internal evolving algebra representation, which is stored in the file `faclist.gea`. When the preprocessor is done, the user is returned to the Gofer prompt.

Step 3: loading an evolving algebra Finally, the preprocessed evolving algebra can be loaded into EVADE. To load the example evolving algebra into the system, the user must type the following at the Gofer prompt:

```
? :a faclist.gea
```

(The Gofer interpreter command `:a` loads the specified file into Gofer *in addition* to the files that have been loaded before.) As a result, the file `faclist.gea` in which the internal evolving algebra representation is stored, is loaded into Gofer in addition to the EVADE program files that were already loaded. The user is presented again with the Gofer prompt, and two new identifiers are now available to him:

- `faclistEA`: the evolving algebra specified by the evolving algebra specification in the file `faclist.ea`.
- `faclist`: the function which results from applying the compiler to the evolving algebra `faclistEA`.

To the first of these two identifiers, the compiler and the run analyzer can be applied. The second identifier is a function with the following definition:

```
faclist n  =  compile (faclistEA n)
```

Hence, this function is the result of applying the compiler to the evolving algebra. As a consequence of the availability of this function, the user never explicitly needs to use the function `compile` to invoke the compiler. In stead, he can resort to the function `faclist`, which invokes the compiler implicitly.

The run analyzer can also be applied to the identifier `faclistEA`. How this is done will be explained in the course of the next section.

6.2.2 Using EVADE on the evolving algebra

By the three steps presented above, EVADE and the preprocessed example evolving algebra are loaded into the Gofer environment. All preparations have now been made to start actually using the example evolving algebra. Basically, it can be used in two ways:

- The compiled evolving algebra can be used as a Gofer function for different purposes.
- The runs induced by the evolving algebra can be analyzed.

We will consider these uses of the evolving algebra in turn.

Using the compiled evolving algebra as a function

The compiled example evolving algebra is the Gofer function `faclist`. It can be used as any other Gofer function for different purposes. For instance, we can evaluate the function at the Gofer prompt for a given argument:

```
? faclist 7
  5040
  (7257 reductions, 13775 cells)
```

Alternatively, we can use the function to define other functions. For instance:

```
comb n r  =  faclist n / (faclist r * faclist (n-r))
```

Like any other Gofer function, `faclist` can be partially applied, and used as argument for a higher order function:

```
facs  =  map faclist [1..]
```

According to this definition, `facs` is the infinite list of factorials of the natural numbers.

Finally, the function produced by compiling an evolving algebra can be used as the interpretation of a static function name in another evolving algebra specification. This implies that evolving algebras can be used as functional modules to build larger evolving algebras. An example of such modularization will be given below in section 6.3.3.

Analyzing the run of an evolving algebra

The evolving algebra `faclistEA` induces a transition tree for every argument. We can analyze such a tree with the run analyzer. To enter the run analyzer for the example evolving algebra with argument 7, the user needs to type:

```
? analyze (faclistEA 7)
```

As a result, a welcome message is shown, as well as the initial state of the evolving algebra, which is the root of the transition tree.

```
------------- EvADE ----------------

Transition : START

  n : 7
  ListElem : {@0}
  i : 0
  head : <function>
  tail : <undef>
  root : @0
  last : @0

Quit/Step/Back/Eval/Until/Refresh >>
```

As can be seen from this example, static sorts are not shown. Dynamic sorts are shown as collections of anonymous elements. These anonymous elements are shown as numbered at-signs. Functions are not printable in general, and are therefore shown as `<function>`.

The user now finds himself at the following prompt:

```
Quit/Step/Back/Eval/Until/Refresh >>
```

This prompt lists the run analyzer commands. The user can issue these commands by typing them in full at the prompt, or by giving only their first letter. Using these commands, the user can navigate through the transition tree, and inspect the states which are its nodes.

Inspecting the current state Each time the run analyzer changes to a new state, the content of that state will be displayed. To inspect dynamic functions and non-dynamic portions of the state, the user can issue the **eval** command. This command will ask the user for an arbitrary expression, evaluate this expression and print its value. For example:

```
Quit/Step/Back/Eval/Until/Refresh >> eval
Expression >> head(last)
Value: 1
Quit/Step/Back/Eval/Until/Refresh >> eval
Expression >> root
Value: @0
Quit/Step/Back/Eval/Until/Refresh >> eval
Expression >> tail(root)
Value: <undef>
Quit/Step/Back/Eval/Until/Refresh >>
```

When the user issues the **refresh** command, the current state will be re-displayed. This command is useful especially after a number of **eval** commands have been issued.

Navigate to another state To navigate through the transition tree, the user has three commands at his disposal: `step`, `until`, and `back`. The `step` command can be used to take a single step down the transition tree, to the successor of the current state.

```
Quit/Step/Back/Eval/Until/Refresh >> step
Transition : step

  n : 10
  ListElem : {@1,@0}
  i : 1
  head : <function>
  tail : <function>
  root : @0
  last : @1

Quit/Step/Back/Eval/Until/Refresh >>
```

The `until` command makes it possible to go several steps down the transition tree. When this command is issued, the user is asked for a stop condition. Then, the run analyzer will descend the transition tree until a state is entered in which the stop condition is satisfied.

```
Quit/Step/Back/Eval/Until/Refresh >> until
Stop condition >> i > 4
Transition : step
Transition : step
Transition : step
Transition : step

Stop condition is satisfied.

  n : 10
  ListElem : {@5,@4,@3,@2,@1,@0}
  i : 5
  head : <function>
  tail : <function>
  root : @0
  last : @5

Quit/Step/Back/Eval/Until/Refresh >>
```

The `back` command is used to return to the state that was previously current.

We briefly summarize the available run analyzer commands:

`step`	Make a transition to a successor of the current state.
`back`	Return to the previously current state.
`eval`	Evaluate an expression in the current state.
`until`	Evolve until a given stop condition is satisfied.
`refresh`	Show the current state anew.
`quit`	End the run analyzer session.

These commands can be issued by typing them in full, or just their first letter.

6.3 Advanced features

In the foregoing section we have demonstrated the basic features of EVADE. In this section, we will show how EVADE handles static function definitions, non-determinism and modular evolving algebras.

6.3.1 Static function definitions

In the example of the previous section, we used the predefined operator * for natural number multiplication. To demonstrate how static function definitions are handled in EVADE, we will modify this evolving algebra specification to import the function `mult` in stead of relying on predefined multiplication. The modified specification is as follows:

```
MODULE facmult (n : N) head(last):N
SS
DS ListElem
SF mult : (N,N) -> N  ==> mult
DF i:N
   head : ListElem -> N
   tail : ListElem -> ListElem
   root : ListElem
   last : ListElem
START i := 0
      NEW e : ListElem WITH
        head(e) := 1
        root    := e
        last    := e .
TRANSITION step
    IF i < n
    THEN i := i + 1
         NEW e : ListElem WITH
            head(e)    := mult (head(last),i+1)
            tail(last) := e
            last       := e .
STOP i = n
```

We assume this evolving algebra specification to be present in the file `facmult.ea`. The arrow `==>` indicates that the static function name `mult` on the left hand side, is interpreted as the Gofer function `mult` on the right hand side. The static function name and the name of the Gofer function need not be the same. Of course, this evolving algebra specification must be supplemented with the Gofer specification of `mult`:

```
mult :: Int -> Int -> Int
mult = (*)
```

Note that the Gofer function `mult` is curried, whereas the evolving algebra static function name `mult` is not. We assume this Gofer function definition to be present in the file `mult.gs`.

To load this evolving algebra into EVADE, the user needs to perform an extra step. As before, he must start EVADE and preprocess the evolving algebra by the following commands:

```
unix-> gofer + EvADE.gp
? preprocess "facmult"
```

The preprocessed evolving algebra can be loaded into EvADE *after* the definition of `mult` is loaded. Thus, the user must type the following two commands at the Gofer prompt:

```
? :a mult.gs
? :a facmult.gea
```

Now the modified evolving algebra is loaded, and ready for use.

6.3.2 Static sort definitions

Up till now we have only used the predefined static sort `N`. We will now give an example of a static sort definition in EvADE. To demonstrate how static function definitions are handled in EvADE, we will modify our previous example specification to import the gofer type `Int` in stead of relying on the predefined sort `N`. The modified specification is as follows:

```
MODULE facmultInt (n : Nat) head(last):Nat
SS Nat    ==>     Int
DS ListElem
SF mult : (Nat,Nat) -> Nat  ==> mult
DF i:Nat
   head : ListElem -> Nat
   tail : ListElem -> ListElem
   root : ListElem
   last : ListElem
START i := 0
      NEW e : ListElem WITH
        head(e) := 1
        root    := e
        last    := e .
TRANSITION step
    IF i < n
    THEN i := i + 1
         NEW e : ListElem WITH
            head(e)    := mult (head(last),i+1)
            tail(last) := e
            last       := e .
STOP i = n
```

This specification is obtained from the previous example by replacing all occurrences of `N` by the static sort name `Nat`. In the static sort clause marked by `SS`, this static sort name is bound to the Gofer type `Int`. This evolving algebra can be used in exactly the same way as the previous one.

6.3.3 Modularized evolving algebras

In stead of defining `mult` as a Gofer function, we can define it as an evolving algebra. Consider the following evolving algebra specification:

```
MODULE mult (n:N, m:N) result:N
SS
DS
SF
DF reg1:N
   reg2:N
   result:N
START reg1 := n
      reg2 := m
      result := 0
TRANSITION step
    IF  reg1 /= 0
    THEN reg1 := reg1 - 1
         result := result + reg2
STOP reg1 = 0
```

Compiling this evolving algebra yields the following Gofer function:

```
mult :: N -> N -> N
mult x y =  compile (multEA x y)
```

Loading this evolving algebra in stead of the Gofer script file `mult.gs` will make `facmult` (as well as its derivate `facmultInt`) a modularized evolving algebra.

```
unix-> gofer + EvADE.gp
? preprocess "mult"
? preprocess "facmult"
? :a mult.gea
? :a facmult.gea
```

This modular evolving algebra can be analyzed and used in the same way as its non-modular counterpart.

6.3.4 Non-determinism

The foregoing examples were all deterministic evolving algebras. As an example of a non-deterministic evolving algebras, consider the following specification:

```
MODULE nondet i:N
SS
DS
SF
DF i:N
START i := 0
TRANSITION up
    IF True
    THEN i := i + 1
TRANSITION down
    IF True
    THEN i := i - 1
STOP False
```

We load this evolving algebra into EvADE as follows:

```
unix-> gofer + EvADE.gp
? preprocess "nondet"
? :a nondet.gea
```

We can investigate the transition tree of this evolving algebra with the run analyzer. To enter the run analyzer the user types:

```
? analyze nondetEA
```

As a result, the run analyzer is entered, and the initial state is shown:

```
------------- EvADE ----------------

Transition : START

  i : 0

Quit/Step/Back/Eval/Until/Refresh >>
```

Due to the non-determinism present in this evolving algebra, the current state does not have a unique successor, but two. When the `step` command is issued, the user is asked to choose between the applicable transitions:

```
Quit/Step/Back/Eval/Until/Refresh >> step
Applicable transitions:
    1. up
    2. down
Pick a number between 1 and 2 (or 0 for random choice) >> 2
Transition : down

  i : -1

Quit/Step/Back/Eval/Until/Refresh >>
```

As indicated by the prompt, the user can delegate the choice between applicable transitions to the random number generator, by typing `0`.

The user can retrace his steps to try alternative routes through the transition graph by using the `back` command:

```
Quit/Step/Back/Eval/Until/Refresh >> back
Return to previous state

  i : 0

Quit/Step/Back/Eval/Until/Refresh >>
```

When the `until` command is issued, the random generator will decide between applicable transition rules at each step:

```
Quit/Step/Back/Eval/Until/Refresh >> until
Stop condition >> i>1
Transition : down
Transition : down
```

At the Unix prompt:

`gofer + EvADE.gp`	Load EVADE.

At the Gofer prompt:

`preprocess "fname"`	Preprocess EA specification in `fname.ea`.
`:a fname.gs`	Load Gofer definitions in `fname.gs` into Gofer.
`:a fname.gea`	Load preprocessed EA specification into Gofer.
`name args`	Run an EA with given arguments.
`compile (nameEA args)`	Run an EA with given arguments.
`analyze (nameEA args)`	Start the run analyzer for an EA.

At the run analyzer prompt:

`step`	Transfer to a successor of the current state.
`back`	Return to the previously current state.
`eval`	Evaluate an expression in the current state.
`until`	Evolve until a given stop condition is satisfied.
`refresh`	Show the current state anew.
`quit`	End the run analyzer session.

Table 6.1: EVADE command summary

```
Transition : up
Transition : up
Transition : up

Stop condition is satisfied.

  i : 2

Quit/Step/Back/Eval/Until/Refresh >>
```

6.4 Summary

EVADE consists of a preprocessor, a compiler and a run analyzer for non-deterministic, modularized, many-sorted evolving algebras. Static function and sorts are specified as Gofer functions and types, and imported into the evolving algebra specification.

Table 6.1 summarizes EVADE's commands. EVADE is loaded by typing `gofer + EvADE.gp` at the Unix command prompt. An evolving algebra is preprocessed by typing `preprocess "filename"` at the Gofer prompt, assuming its specification resides in the file `filename.ea`. A preprocessed evolving algebras is loaded into Gofer by typing `:a filename.gea` at the Gofer prompt. As a result, the evolving algebra and its compiled form become available as `nameEA` and `name` in the Gofer environment. The latter function can be used in Gofer definitions and at the Gofer prompt just as any other Gofer function. A run analyzer session for the evolving algebra is started by typing `analyze (nameEA arguments)` at the Gofer prompt.

At each moment during a run analyzer session, one of the states of the transition graph induced by the evolving algebra is the current state. At the start, the initial

state is the current state. The command `step` allows the user to descend the tree one step at a time. The command `until` allows the user to descend several steps in succession, until a state is reached in which a given stop-condition is true. The command `back` allows the user to retrace his steps, to the state that was current before. The command `eval` allows the user to evaluate an expression in the current state. The command `refresh` redisplays the current state. The command `quit` brings the user back at the Gofer prompt.

EVADE satisfies the minimum conditions formulated at the end of chapter 5. It comprises a compiler and a run analyzer that can be used in combination. Also, it accepts many-sorted evolving algebras with functional modules. Dynamic sorts and non-determinism are supported by EVADE as well. Through the functional programming language Gofer, static functions and sorts can be specified elegantly.

In order to obtain a clearer impression of the merits and demerits of EVADE, a comprehensive comparison must be made with other evolving algebra tools. This will be done in the next chapter.

Chapter 7

Comparison of EVADE with other tools

7.1 Comparison with respect to the evolving algebras that are accepted. 93
7.2 Comparison of tools within their categories 96
7.2.1 Comparison of compilers 96
7.2.2 Comparison of run analyzers 96
7.3 General evaluation 98

Evolving algebra tools can be compared from two angles. Firstly, they can be compared with respect to the evolving algebras accepted by them. For instance, some tools may accept non-deterministic evolving algebras, while others only accept deterministic ones. Secondly, tools can be compared within their category, with respect to the specific task they are supposed to perform. For example, some evolving algebra run analyzers may allow the user to trace a run in backward order, while others may allow only forward tracing. We will submit EVADE to comparisons with other tools from both angles.

EVADE comprises a compiler and a run analyzer. We will restrict our comparison from either angle to evolving algebra tools from these two categories. The tools concerned are:

- The modular and non-modular version of lean*EA*.
- The DASL-ALMAcompiler, or DASL-compiler for short.
- The C run analyzer.
- The Scheme run analyzer.

In section 7.1 these tools are compared to EVADE with respect to the evolving algebras that are accepted by them. In section 7.2 we will compare these tools to EVADE within their respective categories. Finally, in section 7.3, a general evaluation of EVADE will be distilled from the various comparisons.

7.1 Comparison with respect to the evolving algebras that are accepted.

Initial state description

For the description of evolving algebra rules, a fairly standard syntactical convention exists. All existing evolving algebra tools follow this convention fairly closely.

For the initial state description, no standard convention is available. In EVADE, the description of the initial state is split into two parts. The first part is a mapping of static functions and sorts to Gofer functions and types. This mapping is integrated into the signature. Naturally, the Gofer functions and types are specified in the language Gofer itself. Gofer types and functions can be specified very elegantly, in a way that is strongly reminiscent of mathematical practise. The second part of the initial state description is the start update set.

All evolving algebra tools on our list, except the non-modular version of lean*EA*, apply such a division of the initial state description. With respect to the start update set, they are all very similar to EVADE. But with respect to the specification of values for static sort names and static function names, significant differences can be noted.

Modular lean*EA* is very similar on this point to EVADE. Integrated into the signature, a mapping is given of static function names to Prolog predicates. These predicates are of course specified in Prolog itself.

The DASL-compiler uses a slightly different setup. To specify the static sorts and functions, no use is made of an external language in which DASL is implemented. In stead, the evolving algebra specification language is extended with a number of constructs with which these specifications can be made. Sorts are defined by giving their constructors, and functions are defined by sets of equations. Thus, mathematical sets and functions are not imported, but specified inside the evolving algebra specification itself.

The C run analyzer offers two ways to give interpretations to static function names. Firstly, C routines can be imported and assigned as interpretations to static function names. This method is not very elegant, because C routines are not necessarily purely functional and do therefore not always specify acceptable mathematical functions. Secondly, static functions can be defined inside the evolving algebra specification itself. In definitions of of functions of arity N, the symbols $\$1..\$N$ must be used as argument names.

The Scheme run analyzer allows Scheme function to be imported as values of static function names.

Many-sorted versus one-sorted evolving algebras

Evolving algebras can be categorized into many-sorted ones and one-sorted ones. In many sorted evolving algebras, the evolving algebra signature specifies a full function signature for every function identifier. This function signature specifies the function to take arguments of certain sorts, and to return a value of a certain sort. If the function is supplied with an argument of inappropriate sort, an error occurs. In one-sorted evolving algebras, the evolving algebra signature specifies only arities of functions, since all arguments and return values are of the same sort. A function can be undefined for arguments of certain sorts, but no sort errors can occur.

EVADE accepts many-sorted evolving algebras. The static sorts of boolean values and integers are predefined. Other static sorts can be imported at will.

The DASL-compiler is many-sorted as well. Static sorts can not be imported into evolving algebras. Instead, the evolving algebra specification is extended with two clauses: one to list new sort names, and one to define constructors on these sorts. These user defined static sorts are allowed to be polymorphic.

The C run analyzer and both the non-modular and the modular version of lean*EA* are one-sorted. The C run analyzer allows the simulation of many sorts by universes.

The Scheme run analyzer has a very unusual attitude towards sorts. Static sorts are not declared explicitly, but inferred from the function signatures. The binding of static sort names to Scheme types is also done implicitly. Since Scheme does not have a strong type system, the static sort system is not strong either. Values may have several types at a time.

Dynamic sorts and universes

In many sorted evolving algebra, the sorts can be divided into static sorts and dynamic sorts. The extend instruction allows dynamic sorts to grow, and the remove instruction allows them to shrink. In one-sorted evolving algebras with universes, the universes can be divided into static ones and dynamic ones. In these algebras, an extend and a remove instruction may also be supported. These allow expansion and contraction of universes in stead of sorts.

EVADE has both static and dynamic sorts. An extend instruction is supported on dynamic sorts. The implementation of support for the remove instruction has been suspended to future extensions of the tool.

Scheme has both static and dynamic universes. It has an extend instruction, but no remove instruction.

The DASL-compiler has both static and dynamic sorts. It has both an extend instruction and a remove instruction on dynamic sorts.

The C compiler has both static and dynamic universes. It has both an extend and a remove instruction on the dynamic universes.

Neither version of lean*EA* has sorts or universes. As a result, no dynamic sort or universes are present, and no extend or remove instructions are supported.

Determinism versus non-determinism

EVADE supports non-deterministic evolving algebras. The compiler chooses between various applicable transition rules with the help of a random number generator. The interactive run analyzer offers the user the possibility to choose between the various applicable rules himself, or to invoke the random number generator.

Non-determinism is not supported by the DASL-compiler, the Scheme run analyzer, the C run analyzer or either of the versions of lean*EA*. When several rules are applicable simultaneously, these tools will always execute the first one encountered.

Modularization

In EVADE, evolving algebra specifications have a functional module header of the kind described in section 3.5. Their parameters are all 0-ary static functions. The EVADE-compiler converts these functional modules to Gofer functions. The resulting Gofer functions can in turn be used as interpretations of static functions in evolving algebra specifications. Hence, EVADE supports functional modules. For an example see section 6.3.3.

The modular version of leanEA likewise supports modules. These modules are syntactically different from the functional modules described in section 3.5 and supported by EVADE. But semantically, they can be classified as functional modules. The headers of these modules have a list of input parameters and a list of output parameters. The leanEA-interpreter converts these modules into two-place Prolog predicates. The first place is intended for a list of actual parameters. The second place is intended for a variable to which the list of output values of the module will be bound. These Prolog predicates can be used to provide interpretations of static functions in evolving algebra specifications.

The DASL-compiler, the C run analyzer and the Scheme run analyzer do not support either functional or procedural modules.

Distributed evolving algebras

The C run analyzer is the only evolving algebra tool that offers support for distributed evolving algebras.

Parallelism

The C run analyzer accepts evolving algebras with variable declarations. Thus, this tool supports massive parallelism. Other forms of parallelism are not supported by any of the existing evolving algebra tools.

External functions

The C run analyzer is the only existing tool that supports external functions.

Stop conditions

An evolving algebra terminates when a state is reached in which no transitions are applicable. EVADE, allows a stop condition to be specified, with which intended and non-intended final states can be distinguished. The termination of an evolving algebra is ruled abnormal when the stop condition is false in its final state (see also section 3.2.2).

The DASL-compiler and the modular version of leanEA also allow a stop condition to be specified, but the semantics of these is different. In leanEA the evolving algebra terminates when the stop condition is true, irrespective of whether any of the transition rules is applicable. In DASL, the stop condition is the guard of a distinguished result-rule, the updates of which are allowed to be output operations only. This rule is fired as soon as the stop condition is satisfied, irrespective of whether any other rules are applicable.

Neither the C run analyzer, nor the Scheme run analyzer accepts evolving algebras with stop conditions. The C run analyzer, however, allows integrity constraints to be imposed on an evolving algebra (see section 7.2.2). With such a constraint, the effect of a stop condition with a semantics as in leanEA and DASL, can be obtained.

7.2 Comparison of tools within their categories

7.2.1 Comparison of compilers

In this subsection we will compare the available evolving algebra compilers: EVADE, the DASL-compiler, and both version of lean*EA*.

Interfaces

The EVADE compiler transforms an evolving algebra into a Gofer function. Naturally, this function takes Gofer values as arguments, and returns a Gofer value. Thus, the EVADE compiler interfaces evolving algebras with Gofer. Since the Gofer environment in turn offers an interface to the operating system, it is possible to give evolving algebras an interface with the operating system.

In the non-modular version of lean*EA*, evolving algebras do not take arguments. Also, they do not return their arguments to their environment, but print them to the screen. Hence, non-modular lean*EA* does not equip its evolving algebras with interfaces.

The modular version of lean*EA* transforms evolving algebras to Prolog predicates. These predicates have two parameters. The first parameter is a list of input values. The second parameter is a list of output values. Hence, lean*EA* provides a logical interface for its algebras.

The DASL-compiler is similar to the non-modular version of lean*EA*. Its evolving algebras take no arguments and print their results in stead of returning them. Hence, these algebras have no interface.

Optimization

The DASL-compiler is the only evolving algebra compiler on our list that applies optimization techniques. The program of the evolving algebra is transformed into a decision tree. The internal node of this tree are conditions that are extracted from the transition rule guards. The leaves of the decision tree are update sets. The compiler also sees to it that common subexpressions become shared. This makes evaluation of these expressions more efficient.

7.2.2 Comparison of run analyzers

In the category of run analyzers, there are three candidates for comparison: EVADE, the C run analyzer, and the Scheme run analyzer. We will perform our comparison on the basis of the user commands offered by these tools.

Execution of transitions

The EVADE-run analyzer offers to commands for descending the transition graph of an evolving algebra: `step` and `until`. The former takes a single step down the tree. The latter descends until a state is encountered in which a given stop condition is satisfied. To let the evolving algebra run until a final state is reached, one can give the condition `False` to the `until` command.

The C run analyzer offers commands with the same functionality. They are called `step` and `run until`. Additionally, the commands `step n`, `run` and `run while` can

be given. The first of these executes a given number of steps, the second runs until a final state is reached and the third runs until a given condition is no longer satisfied.

The Scheme run analyzer offers the commands `run`, which performs one step, `run n`, which performs a given number of steps, and `run 0`, which runs until a final state is reached.

Inspection of states

EVADE offers two commands that serve to inspect the current state: `refresh` and `eval`. The former shows the content of the entire current state. Of course, functions are not printable, and are shown as `<function>`. The `eval`-command can be used to inspect the value of an arbitrary expression in the current state. For instance, the value of a function name for certain arguments can be inspected using `eval`.

The C run analyzer offers a command similar to EVADE's `eval`, which is called `value`. In addition, there is a command called `show` which can be used to inspect various internal data structures of the run analyzer, such as tables of transition rules, and currently defined integrity constraints. The function `show functions` can be used to inspect the entire current state.

The Scheme run analyzer offers a more powerful variant of `eval` and `value`. The command called `scheme` can be used to evaluate an arbitrary scheme expression. Since evolving algebra expressions are modelled as Scheme expressions, this includes the e-valuation of arbitrary evolving algebra expressions in the current state. To show the content of the current state, the Scheme tool offers the command `bindings`. Additionally, there are three commands called `defstat`, `records`, and `runstat`, that result in the display of a very comprehensive list of definition statistics and run-time statistics. These numbers include totals of updates executed, totals of guards evaluated, totals of universe extensions performed, and a ratio that indicates the degree of indeterminism.

Support for integrity constraints

The C run analyzer offers commands to declare, disable and enable integrity constraints. These commands are called `watch`, `enable` and `disable`. When an integrity constraint is enabled, the run analyzer will halt when it reaches a state in which this constraint is not satisfied. Integrity constraints are particularly useful when many-sorts are simulated by universes within a one-sorted evolving algebra. The constraints can than be used to watch the integrity of the simulated sorts.

Backtracking capabilities

EVADE offers the command `back`, which allows the user to return to previous states. This command is especially helpful for the analysis of non-deterministic runs. It avoids the need to rerun the evolving algebra from the initial state in order to explore alternative paths through the transition graph. The C run analyzer and the Scheme run analyzer lack this feature. In stead, they offer commands called, `restart` and `reset`, which unload the current evolving algebra and propels the run analyzer in a state of *tabula rasa*.

7.3 General evaluation

EVADE's weak points

There are a number of features that are supported by some of the evolving algebra tools, but not by EVADE. Some of these lacking features are somewhat controversial from a theoretical standpoint. For instance, the theory of external functions and distributed evolving algebras has not yet been satisfactorily developed. As a result, the lack of support for these features by EVADE need not be viewed as a deficiency. There are also some features lacking in EVADE that have very little added value, such as a run analyzer command for executing a fixed number of transitions.

However, there are a number of desirable features for which support is lacking in in EVADE. These features are:

- Generation of run-time statistics.
- The remove instruction.
- Macros.
- Massive parallelism.
- Integrity constraints.

Since EVADE is many-sorted, integrity constraints are not needed to watch the integrity of simulated sorts. Still, integrity constraints might be useful to test tentative invariants on evolving algebra states. All of the features listed above can be added to EVADE with reasonable little programming effort, and without drastic changes to existing code.

EVADE's strong points

There are a number of desirable features that are lacking from some or all evolving algebra tools, but not from EVADE. These include:

- Functional modules, and an interface to a functional programming language.
- Non-determinism.
- Many-sorts.
- Dynamic sorts.
- Elegant support for the specification of statics.
- Backtracking.

EVADE is unique in supporting all of these features. An additional strong point of EVADE, is that it offers both a compiler and a run analyzer in a single environment. Also, the existence of a separate preprocessor, which is shared by the compiler and the run analyzer, makes it feasible to add more evolving algebra tools to the EVADE environment without too much effort.

In this chapter and the foregoing two, the standpoint of a user was adopted towards evolving algebra tools. In the upcoming three chapters, we will shift our viewpoint. In these chapters, we will describe the implementation of EVADE.

Chapter 8

The programming language Gofer

8.1 Functions and function definitions 100
8.2 Expressions 101
8.3 Types and type definitions 102
8.3.1 Type polymorphism 102
8.3.2 Type constructors 103
8.4 Type classes, overloading and polymorphism restriction 104
8.4.1 Type classes 104
8.4.2 Qualified types, instances and classes 105
8.4.3 Varieties of polymorphism 107

This chapter is the first of a group of three that describe the implementation of EVADE. The program language used in the implementation is surveyed in this chapter. In the upcoming chapters, the monadic programming method used in the implementation and the implementation itself are presented.

The programming language used in the implementation is Gofer. We will give an overview of this language, which is not exhaustive. A complete presentation of Gofer can be found in the user manual of the Gofer system [Jon], on which this section is based.

The following remarks constitute an extremely concise characterization of the language. Gofer is a lazy functional programming language. It is strongly typed, it supports type polymorphism, and its functions are curried. Gofer has a powerful system of type classes, qualified types and overloaded functions.

Gofer was originally developed as an extended subset of the programming language Haskell [HJe92], and is similar to Haskell in many respects. For the benefit of the reader familiar with Haskell, we list the most important improvements of Gofer over Haskell:

- Gofer type classes can take multiple parameters.

- In Gofer, type classes can be instantiated for arbitrary non-overlapping types.
- Gofer contexts may contain arbitrary type expressions.

The new Haskell definition (version 1.3) has adopted most of Gofer's improvements over the previous version of Haskell.

8.1 Functions and function definitions

Gofer functions are defined by sets of functional rewrite rules, called function bindings. In Gofer, functions are "first class citizens". All functions are curried. Functions can be of polymorphic types. The programmer is not compelled to supply type declarations of his functions, but when he does, they are checked to be in accordance with the types inferred by the Gofer system.

Consider the following example of a function definition in Gofer:

```
fac    :: Int -> Int
fac 0  =  1
fac n  =  n * (fac (n-1))
```

The first line of this function declaration is a *type declaration*. It declares the function `min` to be of type `Int -> Int`. Type declarations are not mandatory. The Gofer type inference system does not depend on them. However, the programmer is encouraged to supply them. They are useful for purposes of documentation, restriction of polymorphism, and overloading resolution. The Gofer system compares the declared type with the inferenced type, and reports any discrepancies. Thus, any discrepancies between intended type and actual type are brought to light.

The remaining lines of the example function definition are functional rewrite rules, called *function bindings*. Their order is significant. They are used top-down in pattern matching. The first matching line is used for function evaluation. Variables may occur only once on each left hand side, so equality of arguments can not be expressed implicitly in the pattern. The right hand side of a function binding may be any expression in which the variables of the left hand side are allowed to appear unbound. As is clear from the example, recursion is allowed.

In Gofer, functions are "first class citizens", i.e. functions can be be used as actual arguments and as return values of higher order functions. The possibility of having functions as return values makes it unnecessary to have functions with more than one argument. A function with several, say n, arguments can be simulated by a higher order function with only one argument, which returns a function with $n-1$ arguments. This is called *currying*. The advantage of curried functions over functions of multiple parameters is that they can be partially applied to render more functions. Thus, an elegant programming methodology is made possible by currying: first specify general purpose functions, then create special purpose functions by supplying arguments to the general purpose functions.

In Gofer there are two kinds of function names: ordinary function names, and operators. Ordinary function names start with a lower case letter. Operators are strings of symbols, like `+`, `&&`, `|||`, and `+->`. Ordinary function names are prefix by default, but they can be turned into infix function names by enclosing them in single quotation marks. Thus, we can write `a 'plus' b` in stead of `plus a b`. Operators are infix by default. They can be turned into prefix operators by enclosing them in parentheses. Thus, we can write `(+) a b` in stead of `a + b`.

8.2 Expressions

The most important way of constructing Gofer expressions is function application. Function application has already appeared in the examples above, and is denoted simply by juxtaposition of arguments to a function name.

In addition to function application, Gofer has several special constructions to build expressions. We list the most important ones:

Conditional expression The conditional expression has the following form:

```
if b
then t
else f
```

where `b` is an expression of type `Bool`, and `t` and `f` are expressions of the same type.

Case expression The case expression has the following form:

```
case e of
  p1 -> e1
     :
  pn -> en
```

where `e` is an expression, the p_i are patterns, and the e_i are expressions of the same type.

Lambda expression The lambda expression is of the following form:

```
\p1 .. pn -> e
```

where the p_i are patterns, and `e` is an expression.

Local definitions There are two forms of expressions that introduce local definitions. The first is the `let` expression, the second is the `where` expression. They have the following forms:

```
let p1 = e1           expr where p1 = e1
       :                           :
    pn = en                      pn = en
in expr
```

where the p_i are patterns, the e_i are expressions, and `expr` is an expression.

List comprehension A list comprehension is an expression such as:

```
[ f a | a <- [1,2,3], odd a ]
```

The first expression after the vertical bar `|` is called a generator. Generators bind elements from a list to variables. The second expressions is called a `filter`. Filters are conditions on variables introduced by generators. List comprehension can contain several generators and filters. Their order is significant, and may influence the efficiency with which a list comprehension can be evaluated.

Gofer expressions are evaluated lazily. Note, however, that the pattern matching involved in function bindings may require partial evaluation of arguments to determine which line of a function definition is to be applied.

Lazy evaluation enables the use of infinite data structures. For instance, one can define:

```
infinite  =  1 : infinite
finite    =  take 10 infinite
```

where `take` is the function that takes the initial segment of a given length of a given list. The evaluation of `finite` will produce a finite list in finite time, even though it is defined in terms of an infinite list.

8.3 Types and type definitions

The types of Gofer functions are described by type expressions. Type expressions are built from type variables, type constructors, and type predicates. Type expressions which contain variables denote polymorphic types. New type constructors can added to the set of predefined type constructors by datatype declarations and type synonym declarations. Type expressions containing type predicates are used to describe qualified types, which will not be discussed in the current, but in the next section.

8.3.1 Type polymorphism

Gofer supports type polymorphism. This means that functions can be defined that operate uniformly on arguments of any type. The types of polymorphic functions are denoted by type expressions that contain type variables. Type variables are identifiers that start with lower case letters.

For instance, the function `map` is of polymorphic type:

```
map            :: (a -> b) -> [a] -> [b]
map f []        =  []
map f (x:xs)    =  f x : map f xs
```

(The colon (`:`) in the last line denotes list construction.) The function `map` can be applied to arguments of any type:

```
map (\x -> x+1) [1,2,3,4]        ==>      [2,3,4,5]
map even [1,2,3,4]               ==>      [False,True,False,True]
map isUpper "Foo"                ==>      [True,False,False]
```

Here we use the arrow `==>` to indicate "evaluates to". The function `map` operates on these arguments of different types according to a single function definition.

Another example of a polymorphic function is the `(.)` operator:

```
(.)        :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x   =  f (g x)
```

The `(.)` operator denotes function composition. It allows us to force right association of function application without cluttering up our expressions with parentheses. For instance, we can write `f.g.h x` instead of `f(g(h x))`. It also allows us in certain cases to avoid lambda expressions. For instance we can write `f.g` instead of `\x -> f (g x)`.

8.3.2 Type constructors

There are two kinds of type constructors in Gofer: datatypes and type synonyms. Both kinds of type constructor can be parameterized. If a type constructor does not have any parameters, it is called a base type.

Type synonyms A new type synonym is defined by a type synonym declaration of the following form:

```
type Name a1 .. an = t
```

where `Name` is the name of the new type synonym, which must start with an upper case letter, the a_i are the parameters of the type synonym, and `t` is a type expression which may contain the parameters a_i. Type synonym declarations are *not* allowed to be recursive. The following are examples of type synonym declarations:

```
type Name = String
type List a = [a]
type List = []
```

The effect of type synonym declarations is that the declared synonym will be treated as an abbreviation for the right hand side.

Datatypes A new datatype is defined by a datatype declaration of the following form:

```
data Datatype a1 .. an = C1 t1 .. tp | .. | Cm u1 .. uq
```

where `Datatype` is the name of the new datatype, the a_i are the parameters of the datatype, the C_i are the names of the constructor functions of the new datatype, and the t_i and u_i are type expressions. The name of the datatype must start with an upper case letter. The constructor functions are either prefixed names starting with upper case letters, or infixed strings of symbols starting with a colon. The names of the constructors need not be distinct form the datatype. Datatype declarations are allowed to be recursive. The following are examples of datatype declarations:

```
data Opt a  =  Def a | Undef
data Tree a =  a :< [Tree a]
```

The datatype `Opt` can be used to represent optional types. The datatype `Tree` provides a way to model general trees.

Predefined type constructors A number of type constructors are predefined in Gofer, as well as a considerable number of functions on the types constructed with them.

`Bool` The boolean type contains two values: `True`, and `False`. The predefined boolean functions include: `(&&)` (conjunction), `(||)` (disjunction), `not` (negation), and `otherwise` (a synonym for `True`).

`Char` This is the type of characters. Its values are denoted by characters enclosed in pairs of elevated commas. The predefined character functions include: `isAlpha`, `isUpper`, `isLower`, `isAlphanum`, which test whether their argument is a letter, an upper case letter, a lower case letter, and alpha-numeric character, respectively.

`[a]` This is the type of lists of elements of type `a`. List construction is denoted by a single colon. The empty list is denoted by `[]`. A particular list can either be denoted by repeated application of the list constructor to the empty list (e.g. `(1:(2:(3:[])))`) or by a special syntax using square brackets and commas (e.g. `[1,2,3]`). The predefined list functions include `head`, `tail`, `length`, `null`, `map`, and `filter`.

`String` This is the type of strings. Its values are denoted by strings of characters enclosed in pairs of double quotation marks, or equivalently by lists of characters.

`Int` This is the type of integers. The predefined integer functions include: `(+)`, `(-)`, `(*)`, `div`, and `mod`.

The empty type `[]` This type has the value `()` as its only element.

`a -> b` This is the function type constructor.

`(a,b)` This is the tuple type constructor. Predefined functions on tuples include: `fst` and `snd`. Type constructors for tuples of more than two members are also predefined.

`Dialogue` This is a special type constructor which is provided to do I/O in Gofer without loss of referential transparency. A number of functions are predefined in Gofer to preform I/O using `Dialogue` in a continuation style. These functions include: `readFile`, `writeFile`, `appendFile`, `readChan`, `appendChan`, `done`, `abort`, and `exit`.

Some of the predefined type constructors are built into Gofer, others are defined in the standard prelude, which is loaded automatically into the Gofer interpreter at the beginning of each session.

There are two important predefined functions in Gofer, which can have any type. The first function is `undefined`. When evaluated, this function will cause abnormal termination of the program. The second function is `error "message"`. It has the same effect as `undefined`, but additionally the error message will displayed.

8.4 Type classes, overloading and polymorphism restriction

Gofer allows overloading. This means that a single function name may be used to denote different functions, depending on the types of its actual arguments. In Gofer, overloading is supported by a system of type classes.

8.4.1 Type classes

Type classes group together the (tuples of) types for which an overloaded function is defined. The (tuples of) types that are members of a type class are called its instances. The following is an example of a type class declaration:

```
class Eq a where
 (==)    :: a -> a -> Bool
 (/=)    :: a -> a -> Bool

 x /= y  =  not (x == y)
```

The header of this class declaration introduces the single parameter type predicate `Eq`. The first two lines of the body of the class declaration introduce the functions `(==)` and `(/=)` and indicate that their type is `a -> a -> Bool` if `a` satisfies the type predicate `Eq`. These functions are called member functions of the class. The last line of the class declaration gives a *default* definition for the second member function in terms of the first.

Once a type class has been declared, declarations can be given of instances of the class. For example:

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False
```

The header of the this instance declaration declares the type `Bool` to be an instance of the type class `Eq`. The body of the instance declaration defines the first member function of the type class. A definition of the second member function does not need to be given, because it is already defined by a default definition in the class declaration.

Note that the operator (==) denotes an equivalence relation which is quite different from convertibility (in the sense of having the same (weak head) normal form).

8.4.2 Qualified types, instances and classes

Every class declaration introduces a type predicate of certain arity. Using type predicates, we can formulate *qualified types*. For instance, using the type predicate `Eq` we can formulate the following qualified type:

```
(Eq a, Eq [b]) => a -> b -> c
```

The expression on the left side of the double arrow is called a *type context*. Type contexts are tuples of type predicate applications.

Qualified type expressions can be used to describe the types of member functions of type classes. The types of the member functions of the type class `Eq` are:

```
(==)  ::  Eq a => a -> a -> Bool
(/=)  ::  Eq a => a -> a -> Bool
```

Qualified type expressions can also be used to describe the type of functions that are defined in terms of overloaded functions. For instance:

```
elem          :: Eq a => a -> [a] -> Bool
x `elem` []      =  False
x `elem` (y:ys)  =  if x==y
                    then True
                    else x `elem` ys
```

Functions defined in terms of overloaded functions are overloaded functions as well. They are said to be implicitly overloaded.

Another purpose for which qualified type expressions can be used is type polymorphism restriction. Consider the following example of a polymorphic function:

```
id    :: a -> a
id x  =  x
```

As it is, this function is applicable to values of any type. We can restrict it to be applicable only to values of types `Bool` and `Int` as follows:

```
class Id a
instance Id Bool
instance Id Int

id    :: Id a => a -> a
id x  =  x
```

where `Id` is a type class without member functions. Defined in this way, the `id` function displays restricted polymorphism instead of general polymorphism.

Contexts can be used, not only to qualify types, but also to qualify classes and instances. Consider the following instance declaration:

```
instance (Eq a, Eq b) => Eq (a,b) where
  (a,b) == (c,d)  =  a==c && b==d
```

In this declaration, the member function `(==)` appears in two different roles. As a function on values of type `(a,b)` it is *defined*, but as a function of values of type `a` or type `b` it is *used*. This second role presupposes that the types `a` and `b` are instances of the type class `Eq`. This is expressed by the context `(Eq a, Eq b)` in the header, which qualifies the instance declaration.

Contexts can also appear in the headers of classes. For example, consider the following class declaration:

```
class (Eq a) => Elem a where
  elem :: a -> [a] -> Bool

  x `elem` []     =  False
  x `elem` (y:ys)  =  if x==y
                      then True
                      else x `elem` ys
```

The function `(==)` which is used in the default member function definition presupposes that the type `a` is an instance of type class `Eq`. This is expressed by the context `Eq a` which qualifies the class declaration. The classes that appear in the context of a qualified class declaration are called superclasses of the qualified class.

The variables that appear in the headers of type classes do not need to be ground types, but may be type constructors of any arity. For instance, consider the following class:

```
class Functor t where
  map :: (a -> b) -> t a -> t b
```

From the type declaration of the member function `map` in this class declaration, one can infer that the variable `t` is a one-parameter type constructor. This variable can, for instance, be instantiated with the list constructor `[]` and with the optional type constructor `Opt` as follows:

```
instance Functor [] where
  map f []      =  []
  map f (x:xs)  =  f x : map f xs
```

```
instance Functor Opt where
  map f Undef    =  Undef
  map f (Def x)  =  Def (f x)
```

Since variables in class headers are not restricted to 0-ary types, type classes are often also called constructor classes in Gofer.

8.4.3 Varieties of polymorphism

As one can gather from this section and the foregoing, Gofer supports three forms of polymorphism. These forms have in common that a single function name is used in applications to different types. The differences between these forms concern the wideness of the range of argument types and the number of function definitions associated to the function name.

overloading In the case of overloading, the range of argument types to which the function may be applied is limited. There are several function definitions associated to the function name, one for each set of argument types to which the function is applicable. Which of these definitions is used is determined by the types of the actual arguments.

restricted polymorphism In the case of restricted polymorphism, the range of argument types is limited as in the case of overloading. But there is only one function definition associated to the function name. This definition is used for every set of actual argument types.

general polymorphism In the case of general polymorphism, the range of argument types to which the function may be applied is unlimited. There is only one function definition associated to the function name.

To illustrate the similarities and differences between the three forms of polymorphism, we will give some alternative definitions of the function `map`.

Consider the following declarations of the type class `Map` and its instantiations.

```
class Map a b where
  map :: (a -> b) -> [a] -> [b]
instance Map Bool Bool where
  map f []      =  []
  map f (x:xs)  =  f x : map f xs
instance Map Int Int where
  map f []      =  []
  map f (x:xs)  =  f x : map f xs
```

The member function `map` of the type class `Map` is an overloaded function. The range of argument types to which it is applicable is limited to those for which an instance of `Map` is defined: boolean functions and lists, and integer functions and lists. For each of these sets of argument types a separate function definition is given.

Since the definitions of the member function in each instance declaration are the same, we can replace them by a default definition in the class declaration.

```
class Map a b where
  map :: (a -> b) -> [a] -> [b]
```

```
  map f []      =  []
  map f (x:xs)  =  f x : map f xs
instance Map Bool Bool
instance Map Int Int
```

Now, the function `map` is no longer overloaded, since there is only one function definition. Thus, overloading has been eliminated in favor of restricted polymorphism. We could have defined `map` as a restricted polymorphic function equivalently as follows:

```
class Map a b
instance Map Int Bool
instance Map Bool Int

map :: Map a b => (a -> b) -> [a] -> [b]
map f []      =  []
map f (x:xs)  =  f x : map f xs
```

The function `map` is no longer a member function of the type class `Map`, but is defined separately. The context `Map a b` in the qualified type of `map` ensures that the range of possible argument types remains limited to those for which instances of `Map` have been declared.

If we remove the context from the type of `map` its restricted polymorphism is transformed into general polymorphism:

```
map           :: (a -> b) -> [a] -> [b]
map f []      =  []
map f (x:xs)  =  f x : map f xs
```

The function `map` is no longer restricted to argument type for which instances are defined, but may be applied to functions and lists of any type.

Gofer's extended type system, supporting three varieties of polymorphism, is highly expressive. Due to this type system, Gofer is very well suited for monadic programming. The description of the monadic programming method given in the next section will make ample use of Gofer's type potentials.

Chapter 9

Programming with monads

9.1 Monads 110
9.1.1 Definition of monads 110
9.1.2 Monads in Gofer 111
9.1.3 Example monads 112
9.1.4 Using monads 114
9.2 Monad extensions 116
9.2.1 Example monad extensions 116
9.2.2 Using monad extensions 120
9.3 Monad transformers 121
9.3.1 Definition of monad transformers 122
9.3.2 Monad transformers in Gofer 122
9.3.3 Correspondence of monad transformers to base monads . . 123
9.3.4 Example monad transformers 124
9.3.5 Using monad transformers 128
9.4 Monad transformers and extensions 129
9.4.1 Example monad transformer extensions 129
9.4.2 Using monad transformer extensions 131
9.5 Monadic parsers 133
9.5.1 The parser monad transformer 133
9.5.2 The parser monad transformer and extensions 134
9.5.3 The parser extension 134
9.5.4 Using the parser monad extension 136
9.5.5 Example 137
9.6 Monadic I/O 138
9.6.1 The I/O monad 139
9.6.2 The I/O monad transformer 139
9.6.3 The I/O extension 140
9.6.4 Interactive I/O 141

In this chapter, an account is given of the monadic programming methodology used in the implementation of EVADE. Monads are programming constructs that allow novel kinds of data hiding. They can be used to create highly structured functional programs. The structure of these programs makes them easily adjustable, and allows reuse of their components. Also, these programs can elegantly incorporate many imperative programming features into a purely functional framework, and can be read in an almost imperative way. As a result, monadic programs are powerful, yet readable and simple to understand.

Section 9.1 explains what monads are in general, and lists a number of specific monads. Section 9.2 shows how monads can be extended with additional functionality. Sections 9.1.4 and 9.2.2 demonstrate by example how monads and monad extensions are used.

Monad transformers are constructors that allow complex monads to be built from base monads. Section 9.3 describes what monad transformers are in general, and lists a number of specific monad transformers. Subsequently, section 9.4 shows how transformed monads are extended with additional functionality, and how this functionality is propagated through repeated transformations. Sections 9.3.5 and 9.4.2 demonstrate by example how monad transformers are used to transform monads, and how extended transformed monads are used.

Sections 9.5 and 9.6 give an account of monadic parsing and monadic I/O, respectively.

Monads and monad transformers are notions that stem from category theory. However, no knowledge of category theory is needed to understand and apply monadic programming methods. A good introduction to monadic programming is given by Wadler in [Wad92]. A somewhat more formal account of monads can be found in [Wad90]. Monad extensions and monad transformers were proposed as programming techniques by Liang, Hudak and Jones [LHJ95]. In the present chapter, the various concepts present in these sources have been accommodated into a single coherent presentation. Also, a number of new concepts have been added. These new concepts include the `unlift` and `insert` functions on monad transformers, which will be explained in section 9.3.

9.1 Monads

In this section we answer the question "What is a monad?". We will first give a general answer to this question, culminating in a definition of monads. Then we will answer the question specifically, i.e. by listing a number of particular monads. Finally, we will show by example how a non-monadic function definition can be transformed to a monadic one.

9.1.1 Definition of monads

A monad is a unary type constructor with associated functions that obey special laws. More specifically, if we have a monad by the name M, we have functions of the following names and types:

$$\begin{aligned} unit &: a \rightarrow M\ a \\ bind &: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \\ fun &: (a \rightarrow b) \rightarrow (M\ a \rightarrow M\ b) \\ join &: M\ (M\ a) \rightarrow M\ a \end{aligned}$$

These functions must be defined in such a way that between them the following relationships hold:

$$\begin{aligned} bind\ m\ k &= join(fun\ k\ m) \\ fun\ f\ m &= bind\ m\ (\lambda a\,.\,unit(f\ a)) \\ join\ m &= bind\ m\ id \end{aligned}$$

In fact, given these equations, it suffices to define only *unit* and *bind*, and to derive *fun* and *join*, or, alternatively, to define only *unit*, *fun*, and *join*, and to derive *bind*. We will opt for the first of these alternatives.

The monad functions must not only be of specific types and stand in particular relationships to one another, but they must also obey a set of three monad laws. We will express these laws in terms of *unit* and *bind*, though they can be expressed in terms of *unit*, *fun*, and *join* just as well.

$$\begin{array}{ll} \text{Left unit:} & (unit\ a)\ \lq bind\lq\ k\ =\ k\ a \\ \text{Right unit:} & m\ \lq bind\lq\ unit\ =\ m \\ \text{Associativity:} & m\ \lq bind\lq\ (\lambda a\,.\,(k\ a)\ \lq bind\lq\ (\lambda b\,.\,h\ b)) \\ & =\ (m\ \lq bind\lq\ (\lambda a\,.\,k\ a))\ \lq bind\lq\ (\lambda b\,.\,h\ b) \end{array}$$

Here we have adopted the Gofer convention of elevated commas to make `bind` usable as an infix function.

Thus, monads are characterized by the functions defined on them, by the relationships between these functions, and by the laws imposed on these functions. We can summarize these characterizations in the following concise definition of monads:

Definition Any unary type constructor M on which the functions *unit* and *bind* are defined in such a way that they obey the three monad laws, is a monad.

Again, this definition is formulated in terms of *unit*, and *bind*, but could as well have been formulated in terms of *unit*, *fun*, and *join*. The type requirements are implicit.

9.1.2 Monads in Gofer

Apart from the monad laws, the definition of monads can be expressed in Gofer by the following constructor class `Monad`:

```
class Monad m where
    unit :: a -> m a
    bind :: m a -> (a -> m b) -> m b

    fun  :: (a -> b) -> (m a -> m b)
    join :: m (m a) -> m a

    m `bind` k     = join (fun k m)
    fun f m        = m `bind` \a -> unit (f a)
    join m         = m `bind` id
```

The relationships between the monad functions appear in this constructor class as default methods. When one declares an instance of this class, one can suffice by defining either only `unit` and `bind`, or only `unit`, `fun` and `join`. The default methods ensure that the remaining functions are thereby defined as well.

The monad laws can be expressed in Gofer-like syntax as follows:

```
Left unit:      (unit a) `bind` k  =  k a
Right unit:     m `bind` unit  =  m
Associativity:  m `bind` (\a -> (k a) `bind` (\b -> h b))
                  =  (m `bind` (\a -> k a)) `bind` (\b -> h b)
```

Note that in this formulation of the monad laws the meaning of the equivalence sign '=' is different from the meaning it has in Gofer. Whereas in Gofer this sign denotes the rewrite relation, in the Gofer-like syntax used to formulate the monad laws, it denotes the reflexive-transitive-symmetric closure of the rewrite relation, i.e. it denotes convertibility.

9.1.3 Example monads

Now it has been laid down what monads are in general, it is time to present a few example monads. From here on we will no longer use mathematical notation, but only Gofer syntax (`'='` denotes definitional equality) and Gofer-like syntax (`'='` denotes convertibility).

The identity monad

To begin with, we define a trivial monad:

```
data Id a = Id a

instance Monad Id where
  unit x          = Id x
  (Id x) `bind` k = k x
```

This monad is called the identity monad, because the type constructor `Id`, the function `unit` and the function `bind` are isomorphic to the identity type constructor, the identity function and plain – albeit postfix – function application, respectively. The effect of this monad is to encapsulate exactly one value of type `a`.

The maybe monad

Our second example of a monad is also fairly simple.

```
data Maybe a  =  Just a | Nothing

instance Monad Maybe where
    unit x            =  Just x

    Nothing  `bind` _  =  Nothing
    (Just a) `bind` k  =  k a
```

Whereas the identity monad always contains exactly one value of type `a`, the `Maybe` monad *may* contain a value, i.e. it encapsulates one value or no value at all.

The list monad

Our third example of a monad is again only slightly more complicated.

```
type List a  = [ a]

instance Monad List where
    unit x      =  [x]

    m `bind` k  =  [ b | a <- m, b <- k a ]
```

The list monad encapsulates not exactly one, or either one or zero, but any finite number of values of type `a`. The `unit` function returns the singleton list. The `bind` function is defined in terms of a list comprehension, which applies `k` to all values `a` encapsulated in monad `m`, and gathers the results in a single list.

The exception monad

The following monad is similar to the `Maybe` monad, but has a useful extra.

```
data Expt e a  =  Fail e | Succ a

instance Monad (Expt e) where
  unit               =  Succ

  (Succ a) `bind` k  =  k a
  (Fail e) `bind` k  =  Fail e
```

Like the `Maybe` monad, the Expt monad can contain one or zero values of type `a`. But in the case of zero encapsulated values, the `Maybe` monad contains nothing at all, whereas the `Expt` monad contains an exception of the type `e`.

The state monad

The state monad is defined as follows.

```
data State s a = State (s -> (s,a))
unState (State x) = x

instance Monad (State s) where
  unit a      =  State (\s -> (s,a))

  m `bind` k  =  State (\s -> let (s',a) = (unState m) s
                             in unState (k a) s')
```

The monad `State` is a *function type* constructor. The function involved maps a state to a pair consisting of a new state and a value of type `a`. Hence, the state monad might more appropriately be called a state transformer monad, but we will stick to the shorter name. The auxiliary function `unState` provides a convenient way of removing the tag `State` during operations on the state monad. Like the `Id` monad, the `State` monad contains exactly one value of type `a`. The `unit` function encloses this value inside an identity state transformer, which maps the old state unaltered to the new state. The `bind` function first applies the state transformer enclosed in its

first argument `m` to an initial state `s`. The resulting value `a` and the new state `s'` are fed to the state transformer enclosed in its second argument `k`. Hence, the `bind` function has the effect of successively applying the two state transformers enclosed in its arguments.

Above, five unary type constructors have been declared to be instances of the `Monad` class, and appropriate member functions have been defined for them. However, these declarations and functions do not guarantee by themselves that the type constructors are indeed monads, i.e. that their monad functions have the correct types, stand in the proper relationships to one another, and obey the three monad laws. These matters must be verified if we are to use these type constructors as monads.

These verifications need not all be done by hand. The inspection of the type correctness of the monad functions can freely be left to the Gofer type checker. Further, reliance on the default methods for `fun` and `join` ensures that the proper relationships hold between the monad functions. Only the verification of the monad laws needs to be done by hand[1]. The proofs are fairly straightforward. They are not left as an exercise to the reader, but they are given in appendix A.

9.1.4 Using monads

In the foregoing subsections it has been explained what monads are in general, and a number of specific monads has been listed. It is now time to consider an example of monadic programming. We will call again upon our old friend the factorial function to serve as an example.

We will start from a non-monadic definition of the factorial function, and transform it into a monadic definition in several small steps. Consider the following definition:

```
fac      :: Int -> Int
fac 0    =  1
fac n    =  fac(n-1) * n
```

This definition is equivalent to the following:

```
fac      :: Int -> Int
fac 0    =  1
fac n    =  let x = fac(n-1)
            in x * n
```

Here, a `let` construction has been inserted. This non-monadic definition can straightforwardly be transformed to the following monadic one:

```
fac      :: (Monad m) => Int -> m Int
fac 0    =  unit 1
fac n    =  fac(n-1) `bind` \x ->
            unit (x * n)
```

The transformation consists in the following changes:

- The result type `Int` has been encapsulated into a monad `m`.
- The return value `1` has been encapsulated into a unit monad.

[1] With some ingenuity, the proofs of the monad laws, once drawn up, can be type checked using the Gofer type checker.

- The `let` construction has been replaced by a `bind` function.
- The return value `(x * n)` has been encapsulated into a unit monad.

In order to make the function definition more readable, we will use the operator `>>=` as as synonym for `bind`, and `return` as as synonym for `unit`.

```
fac     :: (Monad m) => Int -> m Int
fac 0   =  return 1
fac n   =  fac(n-1)       >>= \x ->
           return (x * n)
```

We can give this monadic definition an almost imperative reading: If the argument is `0` then return the value `1`, else assign the factorial of `(n-1)` to the variable `x` and return `x * n`. In the sequel, we will also use the operator `>>`. With this operator the expression `m >>= _ -> n`, can be abbreviated to `m >> n`.

In general, a non-monadic function definition can be changed into a monadic one by first enclosing the return type by a monad, then embedding calls to monad-valued functions into `bind` constructions, and finally enclosing all return values by insertion of `unit` functions.

Note that the monadic factorial function is restricted polymorphic. Its type declaration does not specify a specific monad, but uses a type variable `m` instead. The type predicate `Monad` restricts the values of `m` to monads.

We will now show how the monadic factorial function is invoked, and to what values it evaluates. When we use the function, we must resolve its restricted overloading (otherwise, the Gofer interpreter will complain). This can be done by calling the polymorpic factorial function from a non-polymorphic function as follows:

```
facM        :: Int -> M Int
facM        =  fac
```

In this definition, the type constructor `M` is a specific monad, not a type variable restricted to monads. For example, we can use the identity monad, the maybe monad or the state monad in the position of `M`:

```
facId       :: Int -> Id Int
facId       =  fac

facMaybe    :: Int -> Maybe Int
facMaybe    =  fac

facState    :: Int -> State Int Int
facState    =  fac
```

The specific monads `Id`, `Maybe`, and `State` have been supplied, so to speak, as values of the type variable `m`.

We can now ask Gofer to evaluate the following expressions:

```
facId 5                  ==>     Id 120
facMaybe 5               ==>     Just 120
unState (facState 5) 0   ==>     (0,120)
```

Thus, depending on the particular monad used, the same function definition produces differently encapsulated values. In general, a monadic function is independent of the monad with which it is used. Consequently, the function and the monad can be modified and extended independently.

The reader might wonder what is gained by making the factorial function monadic, since it makes no use of the extra's offered by the monads with which it is used. In fact, as yet we do not even have the means for using these extras. The `unit` function allows us to create monads. The `bind` function allows us to produce new monads from old ones. But a quick inspection of the first two monad laws informs us that, starting from unit monads, the `bind` function can only produce more unit monads. Thus, by means of these two monad functions alone, we can never produce non-unit monads. Clearly, we will need more functions, in addition to the standard ones, in order to make use of the extras offered by monads. In the next section, we will introduce these functions.

9.2 Monad extensions

The standard monad functions, `unit`, `bind`, `fun`, and `join`, suffice to make full use of the monad only in case of the trivial identity monad. On the other monads, which have non-unit forms as well as unit-forms, more functions need to be defined besides the standard ones in order to capitalize on the features that set them apart from the trivial monad. These extra functions will extend the functionality of the monads on which they are defined. We will group the extra functions in type classes which we will call monad extensions.

9.2.1 Example monad extensions

In the following, we will declare several monad extensions, and define instances of them for some of the monads given in the previous section. Some of the monad extensions given here are mentioned in passing in [LHJ95].

The zero extension

We can define a `zero` function on monads. Monads with this function defined on them will be called zero-monads.

```
class Monad m => Monad0 m where
    zero    ::  m a
```

A zero-monad must obey the following zero laws:

Left zero: `zero ‘bind‘ k = zero`
Right zero: `m ‘bind‘ (\a -> zero)  =  zero`

The `zero` extension makes it possible to create a distinguished monad of a form which is distinct from the unit form. From the two `zero` laws one can glean why this distinguished monad is called a *zero*: it “nullifies” any monad to which it is bound.

A zero can not sensibly be appointed for every monad. In case of the identity monad `Id`, for instance, all monads have unit form. Since zeros and units must be distinct, no zero can be appointed among the identity monads.

In case of the state monad `State`, there is no such lack of non-unit monads. However, all these non-unit monads encapsulate exactly one value of some type `a`. Since the `zero` function has no arguments of this type, it is not possible to supply the value to be encapsulated by the zero. Hence, no `zero` function can be defined for the `State` monad.

In case of the `Expt` monad, non-unit monads are abundant. None of these encapsulates a value of type `a` — they are all of the form `Fail e`. Hence, they all qualify as candidates to become the distinguished zero. However, any choice among them would necessarily be arbitrary; none of the candidates is more suitable to become a zero than any other. For this reason, it would be awkward to define a zero extension for the exception monad.

None of these obstacles occur for our remaining monads: the `Maybe` monad and the `List` monad. They both have a unique non-unit monad which contains zero values of type `a`: `Nothing` and `[]`, respectively. Hence, we will define zero extensions for both these monads.

```
instance Monad0 Maybe where
    zero            =  Nothing

instance Monad0 List where
    zero       =  []
```

These instance declarations and function definitions do not suffice to establish that `Maybe` and `List` are indeed zero-monads. It needs to be verified that the declared functions are of the proper type, and that they obey the zero laws. The former can be verified by the Gofer type checker. The proofs of the latter have been produced by hand, and can be found in appendix A.

The availability of a zero for certain monads allows us to define a `filter` function on them.

```
filter       :: Monad0 m => (a -> Bool) -> m a -> m a
filter p m   =  m `bind` \a ->
                if p a
                then (unit a)
                else zero
```

This function filters out those monads that encapsulate a value of type `a` which satisfy the given predicate `p`. Monads that do not satisfy the predicate are "nullified".

We can also define a function `sentinel` on monads.

```
sentinel       :: (Monad0 m) => Bool -> m ()
sentinel True  =  nop
sentinel False =  zero
```

The function `nop` ('no operation') is a synonym for `unit ()`. The sentinel function takes a boolean argument, and returns a unit monad or a zero monad depending on whether the argument is `True` of `False`. This function can be used to impose a condition on the evaluation of a monad. Consider the following program fragment:

```
sentinel (x > 0) >>
m
```

Recall that `n >> m` abbreviates `n >>= _ -> m`. Due to the sentinel, the monad `m` will be evaluated only if the condition `(x > 0)` is satisfied. Otherwise, the zero monad will be returned.

The plus extension

We can define a plus function (denoted by the infix operator `|||`) on monads. Monads with this function defined on them will be called plus monads.

```
class Monad0 m => MonadPlus m where
    (|||) :: m a -> m a -> m a
```

The plus function is sometimes called "alternation".

A plus monad must obey the following plus laws:

```
zero ||| m     =  m
m    ||| zero  =  m
(a = zero  /\  b = zero) iff (a ||| b  =  zero)
```

As can be seen from the header of the type class declaration as well as from the plus laws, the plus extension presupposes the zero extension. Thus, it is only sensible to extend those monads to plus monads that have already been extended to zero-monads. Thus, in our case, only the he `Maybe` monad and the `List` monad qualify. They are extended as follows:

```
instance MonadPlus Maybe where
    Just x  ||| y  =  Just x
    Nothing ||| y  =  y

instance MonadPlus List where
    (|||)  =  (++)
```

The Gofer type checker establishes that these functions are of the proper type. In appendix A the proofs are given that these extensions obey the plus laws.

The exception extension

We can define two functions `raise` and `handle` on certain monads, that enable us to raise and handle exceptions. Monads with these functions defined on them are called exception-monads.

```
class Monad m => ExptMonad e m where
    raise  :: e -> m a
    handle :: ((e -> m a), m a) -> m a
```

The function `raise` takes an exception as argument and raises it. The function `handle` takes a pair whose first element is a function, called exception handler, which maps an exception to a monad. The second element of the pair is a monad. If an exception is raised in this last monad, the function `handle` applies the exception handler to the exception.

The monad most suited to be extended to an exception-monad is, not surprisingly, the monad `Expt`. It is extended as follows:

```
instance ExptMonad e (Expt e) where
    raise e              =  Fail e
    handle (h,(Succ a))  =  Succ a
    handle (h,(Fail e))  =  h e
```

Other monads can be extended to become exception-monads. Only those monads which have non-unit forms are eligible. For instance the `Maybe` monad and the `List` monad can be extended as follows:

```
instance ExptMonad () Maybe where
    raise ()             =  Nothing
    handle(h,Nothing)    =  h ()
    handle(h,Just a)     =  Just a

instance ExptMonad () List where
    raise ()      =  []
    handle(h,[])  =  h ()
    handle(h,as)  =  as
```

Both these extensions result in somewhat degenerate exception-monads. The empty type `()`, which has `()` as its only element, serves as exception type in both cases. Thus, there is only one exception that can be raised, and no information can be transmitted by it.

The state extension

On monads that have an internal state, such as the monad `State`, we can define the functions `inspect` and `update`, which inspect and update the internal state, respectively. Monads with these functions defined on them are called state-monads.

```
class Monad m => StateMonad s m where
    update   :: (s -> s) -> m s
    inspect  :: m s
    inspect  =  update id
```

The argument of `update` is a function which maps the previous state to the next. Apart from becoming the internal state, the new state is encapsulated as the value of the resulting monad. The function `inspect` can simply be defined as the update which transforms the internal state to itself, i.e. leaves it unaltered, and encapsulates the unaltered state as value. Thus, `inspect` is by default defined in terms of `update`, and instance declarations need only provide a definition of the latter.

The monad `State` can be extended to become a state-monad as follows.

```
instance StateMonad s (State s) where
    update f  =  State (\s -> let s' = f s
                              in (s',s'))
```

According to this definition the function `f` is applied to the previous state. The state thus obtained is encapsulated both as the next state of the resulting monad and as its value.

9.2.2 Using monad extensions

To illustrate the use of monad extensions, we return to the example of section 9.1.4. Recall the monadic definition of the factorial function:

```
fac     :: (Monad m) => Int -> m Int
fac 0   =  return 1
fac n   =  fac(n-1)       >>= \x ->
           return (x * n)
```

We demonstrated the this definition could be used with different monads as instantiations of `m`. However, the monadic features of these monads were not exploited by the factorial function.

We can now proceed to enhance the functionality of the monadic factorial function to by adding some of the monad extensions. For instance, we can place a sentinel in the function, to detect negative arguments to the factorial function.

```
fac     :: (Monad0 m) => Int -> m Int
fac 0   =  return 1
fac n   =  sentinel (n>0) >>
           fac(n-1)       >>=  \x ->
           return (x * n)
```

If we define `facMaybe` as in section 9.1.4, we can ask Gofer to evaluate the following expressions:

```
facMaybe 5               ==>     Just 120
facMaybe (-5)            ==>     Nothing
```

Whereas the original non-monadic function `fac` would produce a run-time error (due to non-termination), the monadic function with the sentinel simply returns a `zero` monad when presented with a negative argument.

Note that this addition of functionality to the factorial function did not require any changes in the existing parts of the function definition. Also, the type of the function remained unchanged. In general, adding functionality to monadic functions does not affect existing code.

Another way to add to the functionality of the factorial function, is to let it keep track of the number of multiplications it performs.

```
fac     :: (StateMonad Int m) => Int -> m Int
fac 0   =  return 1
fac n   =  fac(n-1)    >>= \x ->
           update (1+) >>
           return (x * n)
```

The internal state of type `Int` is updated by incrementing it by one. If we define `facState` as above, we can ask Gofer to evaluate the following expression:

```
unState (facState 5) 0    ==>     (5,120)
```

The first element of this pair is the multiplication count. The second element is the factorial of `5`.

These two separate extensions of the factorial function can also be combined:

```
fac      :: (Monad0 m, StateMonad Int m) => Int -> m Int
fac 0    =  return 1
fac n    =  sentinel (n>0) >>
            fac(n-1)       >>= \x ->
            update (1+)    >>
            return (x * n)
```

Thus, both the sentinel and the multiplication counter are present in this definition of the factorial function. From the type declaration we can deduce that the type variable `m` is restricted to monads that are both zero-monads and state-monads.

Unfortunately, none of the monads defined in the foregoing are both zero-monads and state-monads. Therefore, we can not yet make use of the factorial function with both a `sentinel` and an `update`. Of course, a monad that is both a zero-monad and a state-monad could be constructed. However, the construction of a new monad would be required each time functionality is added, and at each step these monads would become increasingly complex. Especially the definition of the standard and non-standard monad functions for these complex monads would become unacceptably laborious. To deflect these problems, a method will be presented in the next section to construct new monads from old ones using monad transformers. Using this method, no redefinition of functions will be needed to construct new monads.

9.3 Monad transformers

Each of the extended monads of the previous section has some particular feature, which distinguishes it from other monads. When we want to combine features of different monads, we need to create a new monad from scratch, and extend it with the desired functionality. Thus, we need to define a new unary data constructor which encapsulates in some way values, data and functions that are required to realize the features we want. Further, we will need to declare the data constructor to be an instance of the monad class, and define the standard monad functions `unit` and `bind` on them. Next, we must, for each extension, declare the data constructor to be an instance of the corresponding monad extension class, and define the additional functions on them. Obviously, this method of combining features of monads is highly laborious.

Combining monad features can be facilitated greatly by the use of *monad transformers*. A monad transformer is a type constructor which takes a monad as argument and yields a new monad. For each base monad described in section 9.1, a corresponding monad transformer can be defined, which captures the same monad features. When a monad transformer is applied to an arbitrary monad, this monad is enriched with the features captured in the monad transformer.

Iterative application of monad transformers to an arbitrary monad results in a monad in which both the features of the arbitrary monad as well as the features of the applied monad transformers are accumulated. Thus, monad transformers serve as basic building blocks with which customized monads can be created. Hence, monad transformers allow different monad features to be combined in a single monad, without demanding new data constructors or new monad functions to be defined.

In this section, we will first explain what monad transformers are in general. Subsequently, we will explain the close correspondence of monad transformers with base monads. Then, we will list the particular monad transformers, corresponding to the base monads of section `Monads`. Finally, we will show how monad extensions are

defined for transformed monads, and we will demonstrate the use of extended transformed monads by example.

Monad transformers are proposed as a programming technique in [LHJ95]. Some of the monad transformers to be presented in section 9.3.4 were mentioned in that article.

9.3.1 Definition of monad transformers

What are monad transformers? A monad transformer is a unary type constructor, which takes a monad as argument and is monad-valued. Thus, if we have a monad transformer T, then for every monad M the application $(T\ M)$ is a monad as well. Also, a number of functions obeying special laws are associated with monad transformers.

The functions that are required to be defined on monad transformers have the following names and types:

$$\begin{aligned} lift &: M\ a \rightarrow (T\ M)\ a \\ unlift &: (T\ M)\ a \rightarrow M\ a \end{aligned}$$

The function $lift$ encapsulates a monad m into a transformed monad tm. The function $unlift$ recovers the original monad m from the transformed monad tm.

The monad transformer functions must obey a set of three monad transformer laws.

Unit lift:	$lift\ (unit_m\ a) = unit_{tm}\ a$
Bind lift:	$lift\ (m\ `bind_m`\ \lambda a\,.\,k\ a) = (lift\ m)\ `bind_{tm}`\ (\lambda a\,.\,lift\ (k\ a))$
Unlift:	$unlift\ (lift\ m) = m$

The subscripts have been added to assist the reader in resolving the overloading involved in these formulas.

Thus, monad transformers are characterized by their kind, by the functions defined on them, and by the laws imposed on these functions. We can summarize these characterizations by the following definition:

Definition Any unary type constructor T which takes monads into monads, and on which the functions *lift* and *unlift* are defined in such a way that they obey the three monad transformer laws, is a monad transformer.

9.3.2 Monad transformers in Gofer

Apart from the monad transformer laws, the definition of monad transformers can be expressed in Gofer by the following construction class `MonadT`:

```
class (Monad m, Monad (t m)) => MonadT t m where
    lift :: m a -> t m a
    unlift :: t m a -> m a
```

`MonadT` is a binary constructor class which expresses that its second argument `m` is a monad, and that its first argument `t` transforms `m` into a new monad `t m`.

The monad transformer laws can be expressed in Gofer-like syntax as follows:

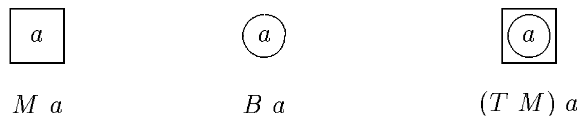
```
Unit lift:  lift (unit a)  =  unit a
Bind lift:  lift (m `bind` \a -> k a)
              =  (lift m) `bind` (\a -> lift (k a))
Unlift:     unlift (lift m)  =  m
```

Note that the names `unit` and `bind` are overloaded, and that the '=' sign denotes convertibility.

9.3.3 Correspondence of monad transformers to base monads

To each base monad which captures certain monad features, a monad transformer corresponds which captures these same features. In fact, in section 9.3.4 we will define each monad transformer in terms of its corresponding base monad.

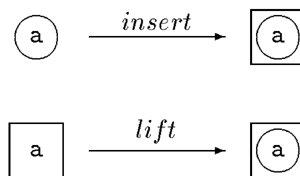
Due to this correspondence, application of a monad transformer to an arbitrary monad yields a monad which can be understood as a combination of the arbitrary monad with the base monad corresponding to the monad transformer. Assume B is a base monad, T is its corresponding monad transformer, and M is an arbitrary monad to which the monad transformer will be applied. Since the monads B and M are each encapsulations of values, the transformed monad $(T\ M)$ can be understood as a two-layer encapsulation. The inner layer is provided by the base monad, while the arbitrary monad envelops the base monad and provides the outer layer. We can depict this situation by the following diagrams:



$M\ a$ $B\ a$ $(T\ M)\ a$

Hence, the monad transformer T combines its corresponding base monad and the arbitrary monad M in an asymmetrical way.

Because a transformed monad is a combination of a base monad and an arbitrary monad, it can be constructed by two distinct methods. The first method starts from a base monad, and adds the arbitrary monad. The second method starts from an arbitrary monad, and adds a base monad. These two methods can be depicted symbolically as follows:



As indicated by the labels of the arrows, the second method is implemented by the function *lift*, which was introduced in the previous section. It is a standard function on monad transformers, and has type $M\ a \rightarrow (T\ M)\ a$. This function operates by squeezing the unit of the base monad **inside** the arbitrary monad which is its argument. The function which implements the first method has been given the name *insert*. Its type is $B\ a \rightarrow (T\ M)\ a$, where B is the base monad corresponding to T. This function operates by wrapping a base monad inside the unit of the arbitrary monad M.

In Gofer, the function *lift* was represented as the member function `lift` of the class `MonadT`. We will represent *insert* as a member function of a class `MonadI`.

```
class (MonadT t m) => MonadI t m b where
  insert :: b a -> t m a
```

This class is a subclass of `MonadT`, as can be seen from the class header. The type predicate `MonadI T M B` expresses that the monad transformer `T` corresponds to the base monad `B`.

As was established above, a transformed monad is an asymmetrical combination of a base monad and an arbitrary monad. Correspondingly, the standard monad functions `unit` and `bind` for the transformed monad are asymmetrical combinations of the standard monad functions for the base monad and the arbitrary monad. The method of combination is very similar to the method for deriving a monadic function from a non-monadic function, demonstrated in section 9.1.4. In that section, we applied the method to transform the non-monadic factorial function of type `Int -> Int` to a monadic factorial function of type `Int -> M Int`. In essence, the method effects the encapsulation of the result values of a function inside an arbitrary monad `M` by introducing the standard monad functions `unit` and `bind` for `M`. When we apply the same method to the standard monad functions `unit` and `bind` for the base method, which have types `a -> B a` and `B a -> (a -> B b) -> B b`, we will obtain the `unit` and `bind` functions for the transformed monad, which have types `a -> T M a` and `T M a -> (a -> T M b) -> T M b`. Thus, in the asymmetrical combination of standard monad functions, the standard monad functions of the base monad play the role of the functions whose result values are encapsulated, and the standard monad functions of the arbitrary monad play the role of the functions that perform the encapsulation. In the upcoming sections, this method will be applied.

9.3.4 Example monad transformers

Now it has been laid down what monad transformers are in general, it is time to present a few example transformers. Each of these captures the same additional functionality as one of the base monads from section `Monads`.

The maybe monad transformer

The maybe monad transformer corresponds to the maybe monad. To define the maybe monad transformer we begin with the definition of the type constructor `MaybeT` in terms of the type constructor `Maybe`:

```
data MaybeT m a       =  MaybeM (m (Maybe a))
unMaybeM (MaybeM m)   =  m
```

The type constructor `MaybeT` is formed by enclosing the constructor `Maybe` inside an arbitrary monad `m` and adding a tag `MaybeM`. Thus, the maybe monad transformer combines the maybe monad and and arbitrary monad in an asymmetrical way. The auxiliary function `unMaybeM` provides a convenient way of removing the tag from the transformed monad, to facilitate the use of the monad transformer.

The definition of the type constructor `MaybeT` alone does not suffice to define the maybe monad transformer. It must be supplemented instance declarations of the three classes `Monad`, `MonadT`, and `MonadI`. The first of these instance declarations asserts that transforming an arbitrary monad by `MonadT` again yields a monad:

```
instance (Monad m) => Monad (MaybeT m) where
  unit a      =  MaybeM (unit (Just a))
  m `bind` k  =  MaybeM (unMaybeM m `bind` \maybea ->
                           case maybea of
```

```
                                   Just a    -> unMaybeM (k a)
                                   Nothing  -> unit Nothing)
```

Note that the names `unit` and `bind` are overloaded. In the left hand side they denote the standard monad functions of the transformed monad. In the right hand side they refer to the standard monad functions of `m`. In the body of this instance declaration, the former are defined in terms of the latter.

These function definitions have been obtained from the standard monad functions for `Maybe` by applying the method of transforming non-monadic functions to monadic ones. Recall the definition of `unit` and `bind` for `Maybe`.

```
unit a              =  Just a

Nothing  `bind` _  =  Nothing
(Just a) `bind` k  =  k a
```

We can rewrite the definition of `bind` as follows:

```
m `bind` k  =  let maybea = m
               in case maybea of
                   Nothing  -> Nothing
                   Just a   -> (k a)
```

By introduction of the `unit` and `bind` for `m`, the return value of these functions can be encapsulated in the monad `m`:

```
unit a      =  unit (Just a)
m `bind` k  =  m `bind` \maybea ->
               case maybea of
                 Just a   -> (k a)
                 Nothing  -> unit Nothing
```

To these definitions we only need to add the tag `MaybeM` and the untagging function `unMaybeM` in the appropriate places to obtain the definitions of `unit` and `bind` for `MaybeT`, as they were given in the instance declaration above.

The second instance declaration that needs to be given declares the type constructor `MaybeT` to be a monad transformer:

```
instance (Monad m, Monad (MaybeT m)) => MonadT MaybeT m where
  lift m     =  MaybeM (m `bind` \a ->
                        unit (Just a))
  unlift tm  =  (unMaybeM tm) `bind` \maybea ->
                case maybea of
                  Just a  -> unit a
                  Nothing -> error "Tried to unlift Nothing"
```

Recall that the task of the `lift` function it to squeeze the unit of the base monad `Maybe` inside the arbitrary monad which is its argument. To perform this task, the `unit` and `bind` of the monad `m` are used. The function `unlift` reverses the effect of the `lift` function, if possible. If not, a run-time error is generated. The third monad law, called 'Unlift' guarantees that application of `unlift` immediately after `lift` is always successful. Tags and untagging functions have been added in appropriate places.

To complete the definition of the maybe monad transformer, we give the following instance declaration:

```
instance MonadI MaybeT m Maybe where
  insert maybea  =  MaybeM (unit maybea)
```

This instance declaration expresses the correspondence of `MaybeT` with `Maybe`. The function `insert` envelops the base monad inside the `unit` of the monad `m`, and tags it with `MaybeM`.

The exception monad transformer

The monad transformer corresponding to the exception monad is the exception monad transformer. The definition of this monad transformer proceeds along exactly the same lines as the definition of the maybe monad transformer. First, a type constructor `ExptT` is defined, and subsequently three class instances are declared for this type constructor.

The type constructor `ExptT` is formed by enclosing the constructor `Expt` inside an arbitrary monad `m` and adding a tag `ExptM`, as follows:

```
data ExptT e m a   =  ExptM (m (Expt e a))
unExptM (ExptM m)  =  m
```

Thus, the exception monad transformer combines the exception monad with an arbitrary monad `m`. The auxiliary function `unExptM` provides a convenient way of removing the tag from the transformed monad.

The first instance declaration asserts that application of the exception monad transformer to an arbitrary monad yields a monad again:

```
instance Monad m => Monad (ExptT e m) where
  unit a      =  ExptM (unit (Succ a))
  m `bind` k  =  ExptM (unExptM m `bind` \expta ->
                         case expta of
                           Succ a -> unExptM (k a)
                           Fail e -> unit (Fail e))
```

The definitions of `unit` and `bind` in this instance declaration are derived from the `unit` and `bind` of the base monad `Expt`. This derivation is exactly analogous to the derivation of the standard monad functions for `MaybeT`.

The second instance declaration asserts that `ExptT e` is a monad transformer.

```
instance (Monad m, Monad (ExptT e m)) => MonadT (ExptT e) m where
  lift m     =  MaybeM (m `bind` \a ->
                        unit (Succ a))
  unlift tm  =  (unExptM tm) `bind` \expta ->
                case expta of
                  (Succ a) -> unit a
                  (Fail e) -> error "Exception was raised, but not handled"
```

The function `lift` squeezes `Succ`, which is the unit of the base monad `Expt` inside the arbitrary monad `m`. The function `unlift` reverses the effect of `lift`, if possible. In case an exception is encountered, unlifting is not possible, and an run-time error is generated.

The final instance declaration for `ExptT` establishes the correspondence between this monad transformer and the base monad `Expt`.

```
instance MonadI (ExptT e) m (Expt e) where
  insert expta  =  ExptM (unit expta)
```

The function `insert` encapsulates the base monad inside an arbitrary monad `m`.

The state monad transformer

The monad transformer that corresponds to the state monad is the state monad transformer. Like the maybe monad transformer and the exception monad transformer, its definition consists of a data constructor definition and three instance declarations.

Recall the definition of the type constructor `State`.

```
data State s a = State (s -> (s,a))
```

Leaving the tag `State` out of consideration, this type constructor forms the basis of the constructor `StateT`:

```
data StateT s m a    =  StateM (s -> m (s,a))
unStateM (StateM m)  =  m
```

Due to the fact that the base monad is a function type in stead of a ground type, this definition is a bit trickier than the definitions of `MaybeT` and `ExptT`. In this case, the arbitrary monad `m` encapsulates not the base monad in its entirety, but only its result type `(s,a)`. This is in complete accordance with the method for transforming non-monadic functions into monadic ones, which was demonstrated in section 9.1. According to this method, the return values and return types of functions must be encapsulated to make them monadic. Hence, the derivation of the monad transformer `StateT` from the base monad `State` is slightly more general, but not essentially different from the derivation of the other monad transformers from base monads.

The following instance declaration asserts that transforming an arbitrary monad by the state monad transformer yields again a monad.

```
instance Monad m => Monad (StateT s m) where
    unit a      =  StateM (\s -> unit (s,a))
    m `bind` k  =  StateM (\s -> unStateM m s `bind` \(s',a) ->
                              unStateM (k a) s')
```

The definitions of the `unit` and `bind` for the transformed monad `StateT s m` are derived from the definitions of these same functions for the base monad `State s`. This derivation is again slightly more general than in the case of the maybe monad transformer and the exception monad transformer. The additional generality is due to the fact that functions in stead of ground values are involved. In accordance of the method of making functions monadic, these functions are not encapsulated in their entirety in the monad `m`, but only their result values.

The second instance declaration for `StateT` asserts that it is indeed a monad transformer:

```
instance (Monad m, Monad (StateT s m)) => MonadT (StateT s) m where
    lift m     =  StateM (\s -> m `bind` \a ->
                              unit (s,a))
    unlift tm  =  (unStateM tm) undefined `bind` \(s,a) ->
                 unit a
```

Again, some additional generality is present in these definitions. The unit of the base monad `State` is the function `\s -> (s,a)` (disregarding the tag `State`). The lift function squeezes not this entire function inside the arbitrary monad `m`, but only its result value `(s,a)`. The unlift function reverses the effect of the lift function. To do this, it supplies an undefined initial state to the transformed monad. Hence, if one tries to unlift a transformed monad which inspects its state before initializing it properly, a run-time error will result.

The definition of the state monad transformer is completed with the following instance declaration:

```
instance MonadI (StateT s) m (State s) where
  insert b  =  StateM (\s -> unit (unState b s))
```

This instance declaration reflects the correspondence of the state monad transformer to the base state monad. The function `insert` envelops the result value of the base monad with the unit of the aribitrary monad `m` (while removing the tag `State`). This instance declaration concludes the definition of the state monad.

9.3.5 Using monad transformers

We will now explain how monad transformers can be used to combine monads into larger monads. In section 9.1.4, the base monad `State Int` and the base monad `Maybe` were used seperately. These monads can be combined into a single monad in four distinct ways. The first way consists in transforming the maybe monad with the state monad transformer:

```
type M  =  StateT Int Maybe
```

The resulting monad `M` provides the state monad as inner encapsulation and the maybe monad as outer encapsulation. The converse transformation consisits in transforming the state monad with the maybe monad transformer:

```
type M  =  MaybeT (State Int)
```

In this case, the maybe monad is the inner encapsulation, while the state monad is the outer encapsulation.

Both these methods of creating a combined monad and maybe monad perform a single transformation on a base monad. Alternatively, we can take the identity monad as base monad, and apply two transformations to it. This can be done in two ways:

```
type M  =  StateT Int (MaybeT Id)
type M  =  MaybeT (StateT Int Id)
```

These two monads are isomorphic to the ones above. The only difference is the extra enveloping encapsulation provided by the identity monad.

From these last two methods of combining monads, one can conclude that one could do away with all base monads except the identity monad. All complex monads are then created by succesive transformations of the base monad. However, these resulting monads have an additional encapsulation which needs to be peeled away and re-installed constantly during monad operations. Therefore, we will keep the base monads to be able to avoid this decrease of efficency.

Monad transformers can be used to combine features of different base monads inside a single complex monad. Unfortunately, we can as yet not use the monad features

present inside these transformed monads. We first need to define the monad extensions corresponding to these features for each of the monad transformers.

9.4 Monad transformers and extensions

On the monads produced by applying monad transformers to monads only the standard monad functions (`unit` and `bind`) and the monad transformer functions (`lift` and `unlift`) are defined. The monad extension functions, such as `zero`, `|||`, `raise` and `update` now need to be defined on them.

9.4.1 Example monad transformer extensions

The zero extension

The zero extension interacts with monad transformers in two ways. Firstly, the maybe monad transformer adds zero functionality to an arbitrary monad. Secondly, other monad transformers propagate this functionality from monads to transformed monads.

The first interaction is expressed as follows:

```
instance (Monad m) => Monad0 (MaybeT m) where
  zero      =  MaybeM (unit Nothing)
```

Thus, transforming a monad by `MaybeT` produces a zero-monad.

The second interaction between the zero extension and monad transformers is expressed as follows:

```
instance (Monad0 m, MonadT t m) => Monad0 (t m) where
    zero       =  lift zero
```

Thus, when a monad transformers is applied to a zero monad, the resulting monad is a zero monad as well. The `zero` function of the transformed monad is defined in terms of the `zero` function of the underlying monad. To produce a transformed zero, it is sufficient to lift the underlying zero.

The plus extension

The plus extension interacts with monad transformers in two ways as well. Firstly, the maybe monad transformer adds plus functionality to an arbitrary zero-monad. This interaction is expressed as follows:

```
instance (Monad m) => MonadPlus (MaybeT m) where
  tm1 ||| tm2  =  MaybeM (unMaybeM tm1 `bind` \x ->
                          case x of
                             Just a  ->  unit (Just a)
                             Nothing ->  unMaybeM tm2)
```

The definition of the member function `|||` is derived from the definition of `|||` for the base monad `Maybe` by making it monadic with respect to `m`.

The plus extension interacts with monad transformers in a second way. When a plus monad is transformed by the state monad transformer or the exception monad transformer, the resulting monad is a plus monad as well. Lifting the plus function through monad transformers can not be done generically for all monad transformers, as was done for the zero monad. Naively, one might propose the following definition:

```
instance (MonadPlus m, MonadT t m) => MonadPlus (t m) where
    tm ||| tn  =  lift ((unlift tm) ||| (unlift tn))
```

However, this definition will not do. The `unlift` functions ruthlessly strip away the encapsulation, and discard any information that is represented by it. After the application of the underlying plus, the `lift` function merely adds a null-layer to the resulting underlying monad. Thus, information is lost in the process of unlifting an lifting. Consequently, to lift the plus function through monad transformers, we can not make use of the monad transformer functions `lift` and `unlift`.

Instead, the lifting of the plus function must be performed non-generically, i.e. seperately for each monad transformer. The appropriate declarations for the monad transformers `StateT` and `ExptT` are as follows.

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
    (StateM x) ||| (StateM y)  =  StateM (\s -> (x s) ||| (y s))

instance (MonadPlus m) => MonadPlus (ExptT b m) where
    (ExptM x) ||| (ExptM y)  =  ExptM (filtr nofail (x ||| y))
                                where nofail (Succ _) = True
                                      nofail (Fail _) = False
```

In these definitions, no use is made of the generic functions `lift` and `unlift` for state monad transformers and exception monad transformers. Thus, the problem of losing information that occured in the naive definition is avoided. Note that it is not sensible to lift the plus extension through the maybe monad transformer. The reason for this is that an underlying plus monad must already have zero functionality, since the zero extension is presupposed by the plus extension. Applying the maybe monad transformer to a monad which already has zero functionality does not add any additional functionality.

The state extension

The interaction of the state extension with monad transformers is twofold. In the first place, when a state monad is transformed by an arbitrary monad transformer, the resulting monad is a state monad as well. This is expressed by the following declaration.

```
instance (StateMonad s m, MonadT t m) => StateMonad s (t m) where
    update f = lift (update f)
```

The `update` function of the transformed monad is defined in terms of the `update` function of the underlying monad. To update the transformed monad, the underlying monad is updated and subsequenlty lifted.

The second interaction of the state extension and the state monad transformer is expressed by the following instance declaration:

```
instance Monad m => StateMonad s (StateT s m) where
    update f = StateM (\s -> unit (s',s'))
               where s' =  f s
```

Thus, the transformation of an arbitrary monad by the state monad transformer results in a tranformed monad that is a state monad. The `update` function of the transformed monad is defined in terms of the standard monad function `unit` of the underlying monad.

The exception extension

The interaction of the exception extension with monad transformers is twofold as well. On the one hand, transformation of an exception monad by an arbitrary monad transformer results in a transformed monad that is an exception monad as well. On the other hand, transformation of an arbitrary monad by the exception monad transformer also results in a monad that is an exeception monad. The first interaction can not be expressed generically, due to problems similar to those that prevented the plus function to be defined generically. This time the `handle` function is problematic. Naively, one could make the following declaration:

```
instance (MonadT t m, ExptMonad b m) => ExptMonad b (t m) where
    raise           =  lift . raise
    handle (f, tm)  =  lift (handle (\b -> unlift (f b), unlift tm))
```

Unfortunately, this declaration will not do. The subsequent unlifting and lifting of tranformed monads will generally result in unacceptable loss of information. Therefore, a generic solution is not available, and the `handle` function will have to be lifted through every monad transformer individually. For the state and exception monad transformers, the appropriate definitions are as follows.

```
instance (MonadT (StateT s) m, ExptMonad b m)
         => ExptMonad b (StateT s m) where
    raise          =  lift . raise
    handle (f, m)  =  StateM (\s -> handle (\b -> unStateM (f b) s,
                                            (unStateM m) s))

instance (MonadT MaybeT m, ExptMonad b m)
         => ExptMonad b (MaybeT m) where
    raise          =  lift . raise
    handle (f, m)  =  MaybeM (handle (\b -> unMaybeM (f b),
                                      unMaybe m))
```

The unproblematic function `raise` is defined in terms of the generic function `lift`. The problematic function `handle` is defined without the help of `lift` or `unlift`.

The second form of interaction between the exception extension and monad transformers is expressed as follows.

```
instance Monad m => ExptMonad b (ExptT b m) where
    raise                  =  ExptM . unit . Fail
    handle (h, (ExptM c))  =  ExptM (c `bind` \a ->
                                     case a of
                                       (Succ x) -> unit a
                                       (Fail y) -> (unExptM . h) y)
```

Thus, when an arbitrary monad is transformed by `ExptT`, an exception monad results. The exception functions `raise` and `handle` of this exception monad are defined in terms of the standard monad functions `unit` and `bind` of the underlying monad.

9.4.2 Using monad transformer extensions

We will now return to the example of the monadic factorial function to demonstrate the use of monad transformers. Recall the last definition of this function:

```
fac     :: (Monad0 m, StateMonad Int m) => Int -> m Int
fac 0   =  return 1
fac n   =  sentinel (n>0) >>
           fac(n-1)       >>= \x ->
           update (1+)    >>
           return (x * n)
```

This definition makes use of the `sentinel` function, which requires `m` to be a zero-monad, and the `update` function, which requires `m` to be a state-monad. To be able to use this definition of the factorial function, we need to construct a monad that is both a zero-monad and a state-monad. In section 9.3.5 four methods to construct such a monad have been demonstrated. In this section, we will use only the first two of these methods.

The `Maybe` monad is a zero-monad. We can add state functionality to it by transforming it with the state monad transformer.

```
type M  =  StateT Int Maybe
```

Using this monad, we can define:

```
facM     :: Int -> M Int
facM     =  fac
```

And ask Gofer to evaluate:

```
unStateM (facM 5) 0       ==> Just (5,120)
unStateM (facM (-5)) 0    ==> Nothing
```

Thus, a non-negative argument `n` result in a monad that encapsulates the number of multiplications and the factorial of `n`. A negative argument results in a zero monad that encapsulates no multiplication statistics.

Alternatively, we can transform the base monad `State` with the `MaybeT` monad transformer. In this way, zero-functionality as added to the state monad which has state-functionality.

```
type M  =  MaybeT (State Int)
```

Defining the function `facM` as above, we can evaluate the following expressions:

```
unStateM (unMaybeM (facM 5)) 0       ==> (5,Just 120)
unStateM (unMaybeM (facM (-5))) 0    ==> (0,Nothing)
```

Hence, a non-negative argument results again in a monad that encapsulates both the answer and the multiplication count. A negative argument results in a zero monad that encapsulates multiplication statistic.

The difference between the two strategies can be described as follows. In the first case, the arisal of a zero annulls the interal state, while in the second case the internal state is not affected by the arisal of a zero.

In the course of this chapter, the following steps were required to design the monadic factorial function:

- Transform the non-monadic function definition to a monadic one according to the standard method of making functions monadic.

- Add a sentinel to the function definition.
- Add an update to the function definition.
- Construct a monad with both zero and state-functionality by transformation of a base monad.

The additions of monadic functionality to the function were all done without altering existing code. The construction of the complex monad did not involve defining any additional monad functions. Hence, the monadic programming method allows functional programs to be organized into highly independent parts.

9.5 Monadic parsers

In this section we will define a parser monad transformer, and a parser monad extension. We will also show how a parser monad transformer can be simulated by a state monad transformer.

9.5.1 The parser monad transformer

First we define the following type constructor and an associated function:

```
data ParserT m a       =  ParserM (String -> m (a,String))
unParserM (ParserM x)  =  x
```

Thus, a monad containing values of type `a` is transformed into a monad which depends on an input string and contains both a value of type `a` and the rest of the input string. The type constructor `ParserT` is very similar to the type constructor `StateT`. In fact, the former is isomorphic to the latter instantiated with `String` as the state type. Due to this isomorphism we will later be able to simulate the parser monad transformer by a state monad transformer.

Second, we make two instance declarations to assert that `ParserT` is a monad transformer:

```
instance (Monad m) => Monad (ParserT m) where
    unit x      =  ParserM (\s -> unit (x,s))
    m `bind` k  =  ParserM (\s0 -> unParserM m s0     `bind` \(a,s1) ->
                                unParserM (k a) s1)

instance (Monad m, Monad (ParserT m)) => MonadT ParserT m where
    lift m     =  ParserM (\s -> m `bind` \x ->
                               unit (x,s))
    unlift tm  =  (unParserM tm) undefined `bind` \(a,s) ->
                  unit a
```

The first of these declarations asserts that application of `ParserT` to a monad indeed results in a new monad. The second declaration asserts that `ParserT` is a monad transformer. To substantiate these assertions, we need to invoke the help of the Gofer type checker to check that all member functions are of appropriate type, and we need to prove that the monad laws and monad transformer laws are respected by the function definitions. These proofs can be found in appendix A.

The definitions of `unit` and `bind` for the monad transformer `ParserT` are completely isomorphic to those for the monad transformer `StateT`.

9.5.2 The parser monad transformer and extensions

The parser monad transformer can be lifted through the monad extensions defined in the foregoing sections. Some of these liftings are generic; others must be performed seperately for each monad transformer.

The zero extension In section 9.4.1 generic liftings of monad tranformers through the zero extension and through the state extension were given:

```
instance (Monad0 m, MonadT t m) => Monad0 (t m) where
    zero       =  lift zero
    iszero tm  =  iszero (unlift tm)

instance (StateMonad s m, MonadT t m) => StateMonad s (t m) where
    update f = lift (update f)
```

These generic liftinsg work for all monad transformers, including the parser monad transformer. They ensure that any zero-monad transformed by the parser monad transformer is a zero-monad, and that any state-monad transformed by the parser monad transformer is a state-monad.

The plus, state, and exception extension Lifting the plus-monads and exception-monads through monad transformers could not be done generically, as was explained in section 9.4.1. Thus, we need to lift these monad extensions through `ParserT` seperately:

```
instance (MonadPlus m) => MonadPlus (ParserT m) where
  (ParserM p) ||| (ParserM q)  =  ParserM (\s -> (p s) ||| (q s))

instance (MonadT ParserT m, ExptMonad b m)
         => ExptMonad b (ParserT m) where
  raise         =  lift . raise
  handle (f,m)  =  ParserM (\s -> handle (\b -> unParserM (f b) s,
                                            (unParserM m) s))
```

The non-problematic function `raise` is lifted generically using the generic function `lift`, which necessitates the apearance of the type predicate `MonadT` in the second instance declaration header. The function `handle` is raised non-generically.

9.5.3 The parser extension

We can define the following functions on some monads:

- `lookahead` Inspects the input string without consuming any input.
- `sat` Parses the first character in the input string if it satisfies a given condition.
- `eatwhile` Parses the longest initial portion of the input string in which all characters satisfy a given condition.
- `initparse` Initializes a parser with an input string.
- `sp` Applies a parser after removing leading spaces from the input string.

- `endofile` Is a parser which succeeds only if the end of the input stream is reached.

A monad with these functions defined on it is called a parser-monad.

```
class (MonadPlus m) => ParserMonad m where
  lookahead :: m String
  sat       :: (Char -> Bool) -> m Char
  eatwhile  :: (Char -> Bool) -> m String
  initparse :: String -> m ()

  sp        :: m a -> m a
  endoffile :: m ()
  sp p      =  eatwhile isSpace >> p
  endoffile =  filter null lookahead
```

The functions `sp` and `endoffile` are defined using default definitions. Hence, instances of this class will only need to provide definitions of the first four functions.

Interaction of the parser extension with monad transformers The parser extension interacts with monad transformers in two ways. Firstly, the transformation of a parser-monad by a monad transformer produces a parser-monad. The lifting of the parser extension through monad transformers can be done generically.

```
instance (ParserMonad m, MonadT t m) => ParserMonad (t m) where
  lookahead  =  lift lookahead
  sat        =  lift . sat
  eatwhile   =  lift . eatwhile
  initparse  =  lift . initparse
```

Secondly, transformation of an arbitrary zero-monad by the parser monad transformer yields a parser-monad.

```
instance (Monad0 m) => ParserMonad (ParserT m) where
  lookahead  =  ParserM (\s -> unit (s,s))
  sat c      =  ParserM (\s -> case s of
                                 (h:ts) -> if c h
                                           then unit (h,ts)
                                           else zero
                                 _      -> zero)
  eatwhile c =  ParserM (\s -> let (xs,ys) = span c s
                               in  unit (xs,ys))
  initparse s = ParserM (\_ -> unit ((),s))
```

The `span` function maps a predicate `p` and a list `l` to the pair of lists of which the first one is the largest initial segment of `l` of which all elements satisfy `p`. The second member of the pair is the remaining segment of the list.

Thus, the parser monad transformer `ParserT` transforms a zero-monad into a parser-monad. But `ParserT` is not the only monad transformer that can have this effect. Due to the similarity of the state monad transformer `StateT` to the parser monad transformer `ParserT`, the latter can be simulated by the former. This is done by taking `String` as the state type of the state monad transformer, and define the parser functions in terms of the state functions `inspect` and `update`.

```
instance StateMonad String m => ParserMonad m where
  lookahead    =  inspect
  sat c        =  inspect   >>= \s ->
                  case s of
                    (h:ts) -> if c h
                              then update (\_ ->  ts) >>
                                   unit h
                              else zero
                    _      -> zero
  eatwhile c   =  inspect   >>= \s ->
                  let (xs,ys) = span c s
                  in update (const ys) >>
                     unit xs
  initparse s  =  update (const s) >> nop
```

Although a seperate parser monad transformer can be simulated by the state monad transformer, we will not discard it, because a dedicated parser monad transformer is more efficent.

9.5.4 Using the parser monad extension

Using the standard and some non-standard monad functions, the auxiliary monad functions and the parser functions, we can construct parsers. We can classify these functions into two categories: *basic parsers*, and *parser constructors*. The basic parsers serve as raw material from which more complex parsers are constructed by the parser constructors. From a given parser the parser constructors build new, more complex, parsers.

The parser functions `lookahead`, `sat`, `eatwhile`, and `endoffile` are basic parsers. The parser functions `sp` and `initparse` are *parser constructors*.

The standard monad function `unit` (or its synonym `return`) is also a basic parser: the parser that returns a given value without consuming any input. The standard monad function `bind` (or its synonym `>>=`) is a parser constructor: from two parsers `p` and `q` it builds a new parser that first performs `p`, followed by `q`.

The non-standard monad function `zero` is a basic parser: the parser that consumes no input and fails. The non-standard monad function `|||` is a parser constructor: from two parsers `p` and `q` it constructs a new parser that succeeds if `p` or `q` succeeds.

The auxilliary monad functions `many` and `many1` are also parser constructors. From the parser `p` they constuct a parser that performs `p` 0 or more times (1 or more in the case of `many1`) and return the results in a list. The auxilliary function `seq` takes a list of parsers, and performs them one after another. The auxilliary monad function `maybe` is a monad transformer that performs the parser `p` and returns its result tagged with `Just` if it succeeds, and returns `Nothing` if it fails. The auxilliary function `nop` is the basic parser that consumes no input and succeeds.

Using these basic parsers and parser constructors, we can build more complex parsers. For instance, the parser `tok` parses a given token:

```
tok      :: (ParserMonad m) => String -> m ()
tok t    =  seq [ sat (c==) | c <- t ]
```

The parsers `digit` and `number` parse a digit and a natural number respecively:

```
number, digit  :: (ParserMonad m) => m Int
number         =  many1 digit `do` foldl1 (\a x -> 10*a+x)
digit          =  sat isDigit `do` \d -> ord d - ord '0'
```

The function `do` is a infix version of `fun`. The parser `ident` parses identifiers consisting of lower case letters:

```
ident     :: (ParserMonad m) => m String
ident     =  many1 (sat isLower)
```

We can also build additional parser constructors. For instance, the parser constructor `parenthesised` performs a parser `p` after parsing an opening parenthesis and followed by parsing a closing parenthesis:

```
parenthesised     :: (ParserMonad m) => m a -> m a
parenthesised p   =  tok "("   >>
                     p         >>= \a ->
                     tok ")"   >>
                     return a
```

Using the parsers and parser constructors defined so far, we can build parsers for a wide range of grammars.

9.5.5 Example

We will now give an example. Consider the following BNF syntax description for arithmetic expressions:

expression → number | (expression) | unairy operator expression | expression binairy operator expression

unary operator → `-`

binary operator → `+` | `-`

Based on this syntax description, the type of a parse tree for such expressions can be designed.

```
data Expression  =  Num Int
                 |  Unary   UnaryOperator Expression
                 |  Binary  Expression BinaryOperator Expression
data UnaryOperator    =  Neg
data BinaryOperator   =  Plus | Minus
```

Using the monadic parser functions, we can construct a parser for our example grammar which produces parse trees of type `Expression`. This can be done as follows:

```
parseExpression  :: PM Expression
parseExpression  =  (number              >>= \int ->
                     return (Num int))
                    |||
                    (parseUnaryOperator  >>= \unop ->
                     parseExpression     >>= \expr ->
                     return (Unary unop expr))
```

```
                     |||
                     (parseExpression     >>= \expr1 ->
                      parseBinaryOperator >>= \binop ->
                      parseExpression     >>= \expr2 ->
                      return (Binary expr1 binop expr2))

parseUnaryOperator  :: PM UnaryOperator
parseUnaryOperator  =  tok "-" >> unit Neg

parseBinaryOperator :: PM BinaryOperator
parseBinaryOperator =  (tok "+" >> unit Plus)
                       |||
                       (tok "-" >> unit Minus)
```

However, this parser is non-terminating for certain expressions. This is due to the left recursiveness present in the third alternative of our example grammer. A simple solution exists to remove this left recursiveness. This solution is offered by the parser operator `|||=`, which is defined as follows:

```
p |||= pc   =  p ||| (pc p)
```

We use this operator to make a subtle modification to our expression parser:

```
parseExpression  =  ((number              >>= \int ->
                       return (Num int))
                      |||
                      (parseUnaryOperator  >>= \unop ->
                       parseExpression     >>= \expr ->
                       return (Unary unop expr)))
                     |||= \p ->
                     (p                   >>= \expr1 ->
                      parseBinaryOperator >>= \binop ->
                      parseExpression     >>= \expr2 ->
                      return (Binary expr1 binop expr2))
```

In this definition, the third alternative is connected with the first two by `|||=` in stead of `|||`. In the third alternative, the left recursive call to `parseExpression` is supplanted by `p`. The result of these modifications is that not the entire expression parser is called recursively, but only the parser for the first two alternatives. As a result, binary operators become right-associative.

9.6 Monadic I/O

In this section, we will discuss monadic input and output. We define an I/O monad and an monad transformer. We present an I/O extension, and extend both the monad and monad transformer by it. The we will discuss the use of the I/O monad and monad transformer for interactive I/O, i.e. processes where output is generated while not all input has yet been received. As it will turn out, the I/O monad is suited for interactive I/O, but the monads created with the I/O monad transformer are not.

9.6.1 The I/O monad

To define the I/O monad we start by creating the following type constructor, and an associated function:

```
data IO a     =  IO (String -> (a,String,String->String))
unIO (IO x)   =  x
```

As can be seen from this definition, the I/O monad contains a function with argument type `String`. This string represents the input to the program. The result type is the triple `(a,String,String->String)`. Here, the first member `a` is the value of the monad. The second member of type `String` represents the remainder of the input. The third member of type `String->String` is a function which maps the rest of the output to the complete output of the program.

We declare this I/O type constructor to be a monad.

```
instance Monad IO where
  unit x      =  IO (\i -> (x,i,\o->o))
  m `bind` k  =  IO (\i0 -> let (a,i1,o1) = (unIO m) i0
                                (b,i2,o2) = (unIO (k a)) i1
                            in  (b,i2,o1.o2))
```

The unit function encapsulates a value of type `a`. Additionally, it transfers the input of the program to the rest of the program without consuming any of it, and it propagates the output of the rest of the program without adding to it. The definition of the `bind` function involves three input streams: `i0` is the input to the entire monad ``m `bind` k``, `i1` is the input which remains after the first monad `m`, and which is subsequently fed to the second monad `k a`. The input stream `i2` contains the input left after the entire operation. There are two output transformations. `i1` is the output transformation effected by the first monad `m`, and `i2` is the transformation effected by the second monad `k a`. The transformation of the entire monad is obtained by composing these two output transformations: `o1.o2`.

9.6.2 The I/O monad transformer

We will now specify an I/O monad transformer. First, we define the following type constructor, and an associated function.

```
data IOT m a    =  IOM (String -> m (a,String,String->String))
unIOM (IOM x)   =  x
```

The following two instance declarations assert that `IOT` is a monad transformer.

```
instance Monad m => Monad (IOT m) where
  unit x       =  IOM (\i -> unit (x,i,\o->o))
  m `bind` k   =  IOM (\i0 -> (unIOM m) i0    `bind` \(a,i1,o1) ->
                              unIOM (k a) i1  `bind` \(b,i2,o2) ->
                              unit (b,i2,o1.o2))

instance (Monad m, Monad (IOT m)) => MonadT IOT m where
  lift m     =  IOM (\i -> m `bind` \x -> unit (x,i,\o->o))
  unlift tm  =  (unIOM tm) undefined `bind` \(a,i,o) -> unit a
```

9.6.3 The I/O extension

We define an I/O extension by the following type class:

```
class (Monad m) => IOMonad m where
  output  :: String -> m ()
  input   :: m String
  write   :: Char -> m ()
  read    :: m Char
  eof     :: m Bool
  newline :: m ()
  readline  :: m String
  readword  :: m String
```

We will briefly discuss the intended meanings of the member functions.

- `output` Puts a string on the output stream.
- `input` Returns the input string in its entirity.
- `write` Puts a single charachter on the output string.
- `read` Returns the first charachter of the input string.
- `eof` Tests whether the end of the input stream has been reached.
- `newline` Puts a cariage return on the input string.
- `readline` Returns the first complete line from the input string.
- `readword` Returns the first complete word from the input string.

Several of these functions could be defined in terms of one of the others.

The monad `IO` can be extended to an I/O monad as follows:

```
instance IOMonad IO where
  output s  =  IO (\i -> ((),i, \o -> s++o))
  input     =  IO (\i -> (i,"",\o->o))
  write c   =  IO (\i -> ((),i, \o -> c:o))
  read      =  IO (\i -> (head i,tail i,\o->o))
  eof       =  IO (\i -> (null i, i,\o->o))
  newline   =  IO (\i -> ((),i, \o -> '\n':o))
  readline  =  IO (\i -> let (line,rest) = span (\c-> c/='\n') i
                         in  (line,tail rest,\o->o))
  readword  =  IO (\i -> let (line,rest) = span (\c-> c/=' ') i
                         in  (line,tail rest,\o->o))
```

The function `span` takes a predicate `p` and a list `l`, and returns a pair of the longest initial segment of `l` in which the elements satisfy the predicate `p`, and the remainder of `l`. We will explain shortly how these functions can be linked up with the I/O system of Gofer.

The extension of the I/O monad transformer `IOT` is very similar to the extension of the I/O base monad:

```
instance Monad m => IOMonad (IOT m) where
  output s  =  IOM (\i -> unit ((),i, \o -> s++o))
  input     =  IOM (\i -> unit (i,"",\o->o))
  write c   =  IOM (\i -> unit ((),i, \o -> c:o))
  read      =  IOM (\i -> unit (head i,tail i,\o->o))
  eof       =  IOM (\i -> unit (null i, i,\o->o))
  newline   =  IOM (\i -> unit ((),i, \o -> '\n':o))
  readline  =  IOM (\i -> unit (let (line,rest) = span (\c-> c/='\n') i
                                in  (line,tail rest,\o->o)))
  readword  =  IOM (\i -> unit (let (line,rest) = span (\c-> c/=' ') i
                                in  (line,tail rest,\o->o)))
```

To obtain this extension from the previous one, all member functions have been made monadic by inserting `unit` functions, and tags.

The I/O extension is generically lifted through monad transformers as follows:

```
instance (IOMonad m, MonadT t m) => IOMonad (t m) where
    output      = lift . output
    input       = lift input
    write       = lift . write
    read        = lift read
    eof         = lift eof
    newline     = lift newline
    readline    = lift readline
    readword    = lift readword
```

Hence, when an I/O monad is transformed, the resulting monad will be an I/O monad as well.

In the next subsection, we will explain in detail how I/O monads can be linked up with Gofer's I/O system. We will focus in particular on *interactive* I/O.

9.6.4 Interactive I/O

When performing interactive I/O it is essential that output can be produced before all input has been received. This allows the user to interact, i.e. it allows him to provide the system with input depending on the output already given.

As it turns out, monads created with the I/O monad transformer will not in general yield programs that can produce I/O before they terminate. The reasons for this will become clear when we investigate an example.

Suppose we create a monad by transforming the `Maybe` monad with the I/O monad transformer:

```
type M  =  IOT Maybe
```

This would allow us to write a program such as the following:

```
program    :: M ()
program    =  output "Password ? " >>
              readline             >>= \pswd ->
              if correct pswd
              then output "Welcome"
              else zero
```

We would like this program to prompt the user with the string `Password ?` before it tries to read from the input stream. Unfortunately, our program could never operate in this sequence. To understand why this is so, expand the type synonym `M`:

```
M  =  IOT Maybe
   =  IOM (String -> Maybe (a,String,String->String))
```

Since `program` is of type `M ()`, it can have one of two forms:

```
IOM (i -> Just ((),i',o->o'))
IOM (i -> Nothing)
```

The second of these forms is a (lifted) `zero`. If our program has the first form, it will produce output. If it has the second form, it will *not* produce any output. So, to determine whether out program produces output, we need to determine whether it has zero-form or not. Looking at the definition of our program, we can see that it depends on the input into the program whether it takes zero-from or not. Hence, it depends on the input of the program whether the program produces output. So, no output can be generated until the input is read. Hence, the prompt `Password ?` will not appear before the password has been typed in by the user. This is obviously not the desired behaviour.

Consider the following alternative definition of `M`:

```
type M  =  MaybeT IO
```

Expanding this definition:

```
M  =  MaybeT IO
   =  MaybeM (IOM (String -> (Maybe a,String,String->String)))
```

So, now our program can take one of the following two forms:

```
MaybeM (IOM (\i -> (Just (), i', o->o')))
MaybeM (IOM (\i -> (Nothing, i', o->o')))
```

The second of these forms is the zero-form. Hence, our program will produce output, whether our program takes zero-from or not. Hence, the prompt `Password ?` can appear before the user types his password. This is the behaviour we desired our program to display.

We will now define a class `InteractiveIO`. With this class we will provide those monads that are suitable for interactive I/O, with the appropriate functions to actually perform input and output through the Gofer type `Dialogue`. The member function `interactive` takes as argument a monad and returns a function which maps an input string to an output string. The member function `dialogue` is defined in terms of `interactive`. It takes a monad as argument, and returns a dialogue.

```
class InteractiveIO m where
  interactive  :: m () -> (String -> String)
  dialogue     :: m () -> Dialogue
  dialogue m   =  readChan stdin exit (\i ->
                  appendChan stdout (interactive m i) exit done)
```

The monad `IO` is suitable for interactive IO. Therefore, we declare the following instance:

```
instance InteractiveIO IO where
  interactive m i  =  let (_,_,ot) = unIO m i
                      in ot ""
```

Note that the use of don't-cares in the local binding is essential. If we write variables for these, the Gofer interpreter will try to evaluate the encapsulated value and the input stream in order to match these variables before it evaluates `ot ""`. As a result, no output will be displayed until all input has been received.

Since monads created with the monad transformer `IOT` are not in general suitable for interactive I/O, we will *not* declare an instance of `InteractiveIO` for these monads.

Interactivity is lifted through monad transformers by the following declaration:

```
instance (MonadT t m, InteractiveIO m) => InteractiveIO (t m) where
  interactive tm i  =  interactive (unlift tm) i
```

Thus, monads created by transforming a monad suited for interactive I/O are themselves suited for interactive I/O as well. The function `interactive` for the transformed monad is defined in terms of the function `interactive` of the underlying monad, and in terms of the function `unlift`.

We can now return to our password program. If we define the monad `M` as `MaybeT IO`, then we can run this little program as follows:

```
? dialogue program
Password ? TheCorrectPassword
Welcome
(155 reductions, 415 cells)
? dialogue program
Password ? AnIncorrectPassword
Tried to unlift Nothing
(108 reductions, 334 cells)
```

In both cases, the password prompt is displayed before the password is typed in by the user. In the first case, the user provides the correct password, the welcome message is displayed, and the program terminates normally. In the second case, the user provides an incorrect password. The welcome message is not displayed. Instead, the program terminates abnormally and an error message `Tried to unlift Nothing` is generated. This error is due to the unlifting of the zero-monad of type `MaybeT IO` by `interactive`.

In this chapter, a comprehensive treatment has been given of the monadic programming method. In the next chapter, this method will play an essential role in the implementation of EVADE.

Chapter 10

EvADE: monadic implementation in Gofer

10.1 Decomposition of EvADE 145
10.1.1 Top level structure 146
10.1.2 Decomposition of the preprocessor 146
10.1.3 Decomposition of the compiler 147
10.1.4 Decomposition of the run analyzer 147
10.1.5 Deeper levels of subcomponents 147
10.2 The monads for the various components 148
10.3 The exception mechanism of EvADE 150
10.4 Implementation of the low level components 151
10.4.1 Parser 151
10.4.2 Generator 153
10.4.3 Evaluator 155
10.4.4 Executor 159
10.4.5 Random number generator 160
10.4.6 History manager 160
10.5 Implementation of the intermediate level components 161
10.5.1 Engine 161
10.5.2 Random descender 162
10.5.3 Interactive explorer 164
10.6 Implementation of the top level components 166
10.6.1 Preprocessor 167
10.6.2 Compiler 167
10.6.3 Run analyzer 168
10.7 Extendability of EvADE 169

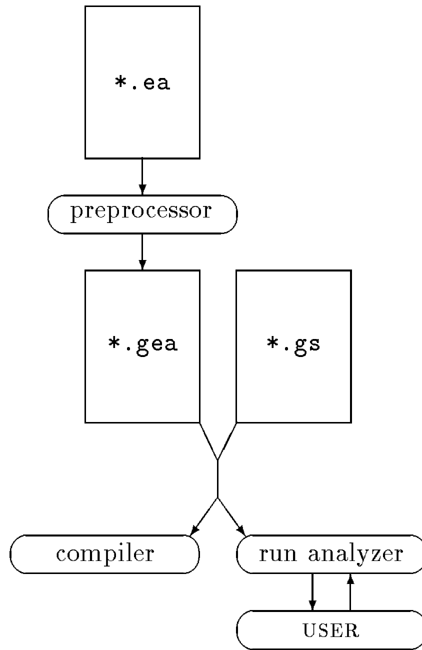


Fig. 10.1: EVADE

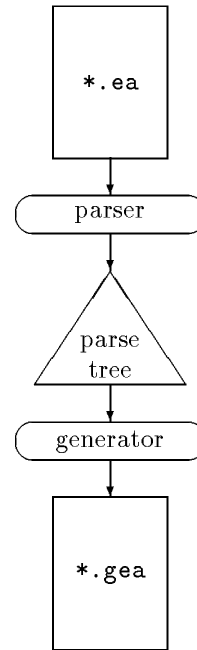


Fig. 10.2: Preprocessor

In chapter 6, the evolving algebra tool EVADE was described from the point of view of a user. In this chapter, the implementation of EVADE will be discussed. EVADE is implemented in the functional language Gofer, and makes ample use of the monadic programming method explained in the previous chapter. Section 10.1 gives an overview of EVADE's internal structure, by decomposing it into components and subcomponents. Section 10.2 introduces the various monads that are used in the implementation of EVADE. In section 10.3 the exception mechanism of the program is explained. The implementation of the low level components of EVADE is discussed in section 10.4. The intermediate level components and top level components are discussed in sections 10.5 and 10.6. Finally, section 10.7 evaluates the implementation of EVADE, and anticipates further development of the tool.

10.1 Decomposition of EVADE

In this section, we will give an overview of the structure of EVADE. First, we will consider the top-level structure in subsection 10.1.1. At this level, three components can be discerned: the preprocessor, the compiler and the run analyzer. In subsections 10.1.2 through 10.1.4 these three components will be further decomposed into subcomponents. In subsection 10.1.5 the decomposition will be carried further still, to culminate in the isolation of six basic building blocks of EVADE.

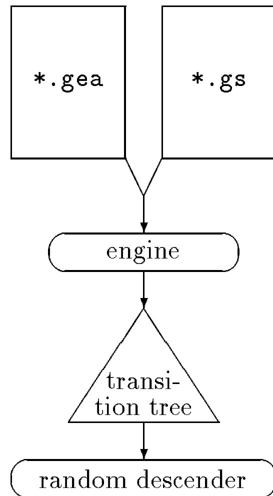


Fig. 10.3: Compiler

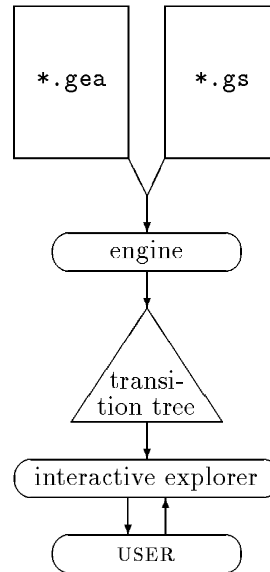


Fig. 10.4: Run analyzer

10.1.1 Top level structure

In chapter 6, the relationship between the three top level components of EVADE and the files used by these components was explained. We will briefly recapitulate this explanation. The accompanying illustration is shown again in figure 10.1.

Evolving algebra specifications are assumed to be stored in files with extension `ea`. EVADE's first component, the preprocessor, converts an evolving algebra specification to an intermediary representation which it stores in a file with extension `gea`. The other two components of EVADE, the run analyzer and the compiler, take the intermediary representation produced by the preprocessor as input. The static functions and sets used by evolving algebra specifications are defined in Gofer script files with extension `gs`. The compiler and the run analyzer take these files as input as well.

We will in turn decompose the preprocessor, the compiler and the run analyzer into subcomponents.

10.1.2 Decomposition of the preprocessor

The task of the preprocessor is to convert an evolving algebra specification to an intermediary representation, which can be read by the compiler and the run analyzer. The preprocessor performs his task in two steps. In the first step, the evolving algebra specification is parsed, resulting in a parse tree. In the second step, the parse tree is stored in a file, together with some additional information extracted from the parse tree. Thus, the preprocessor has two subcomponents: a parser and a generator. The relationships between these subcomponents, the internal parse tree and the files used by the preprocessor is depicted in figure 10.2.

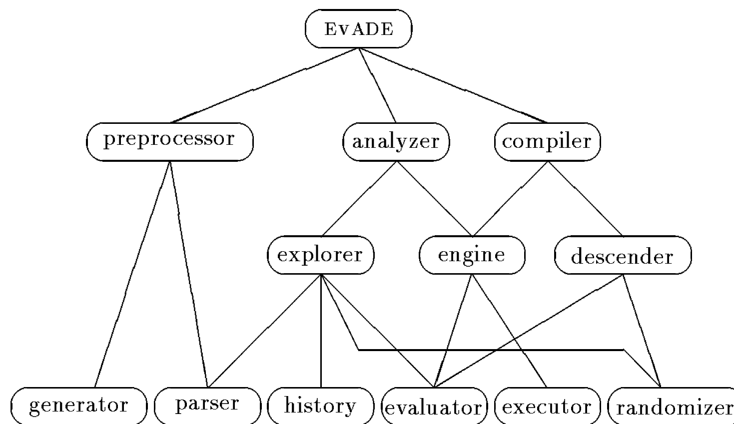


Figure 10.5: Complete decomposition of EVADE

10.1.3 Decomposition of the compiler

The task of EVADE's second component, the compiler, is to descend in a non-determinate way the transition tree induced by an evolving algebra, until a final state is encountered, and to evaluate the return expression of the evolving algebra in that final state. This task of the compiler can be split into two subtasks. The first subtask is to generate the transition tree induced by an evolving algebra. The subcomponent which performs this task is called the *engine*. Since the transition tree of an evolving algebra can be of infinite size, the engine must *lazily* generate this tree.

The second subtask of the compiler consists in following a path through the transition tree until a final state is reached, and evaluating the return expression of the evolving algebra in this state. At each fork along the path, a non-determinate choice must be made among the possible transitions. The resulting path is a non-determinate run of the evolving algebra. The subcomponent that performs this subtask is called the *random descender*. The relationships between the subcomponents of the compiler, the transition tree and the input files are shown in figure 10.3

10.1.4 Decomposition of the run analyzer

The task of the run analyzer is to enable the user to interactively explore the transition tree induced by an evolving algebra. This task is distributed over two subcomponents. The first of these is the engine which is also present in the compiler. It lazily generates the transition tree induced by an evolving algebra. The second component is called an *interactive explorer*. It takes commands and other input from the user, and responds by moving through the transition tree and displaying information. Figure 10.4 graphically presents the relationships between these subcomponents, the transition tree, the input files, and the user.

10.1.5 Deeper levels of subcomponents

From the decompositions given in the previous subsections, one can glean that the three components of EVADE are decomposable into five distinct subcomponents: a parser, a generator, an engine, a random descender, and an interactive explorer. The last three of these subcomponents can be decomposed even further.

Engine The engine has two subordinates: an evaluator and an executor. The executor fires transition rules at evolving algebra states, and delegates the evaluation of conditions and terms to the evaluator.

Descender The random descender invokes two helpers: a random number generator and the same evaluator as was used by the engine. In this case the evaluator is used to evaluate the return expression of an evolving algebra.

Explorer The interactive explorer invokes four subordinates. It uses the parser, which it shares with the preprocessor, to parse user input into shell commands and evolving algebra expressions. Like the engine, it delegates the evaluation of expressions to the evaluator. It shares the random number generator with the descender. Finally, it uses a history manager to archive and recall previous states.

The complete decomposition of EVADE is depicted in figure 10.5. The lowest level of decomposition contains six ultimate subcomponents: the parser, the generator, the evaluator, the executor, the random number generator and the history manager. Hence, these six subcomponents are the basic building blocks from which EVADE is built.

In the upcoming sections we will discuss EVADE's components of different levels in turn, starting at the lowest level. But first, we need to make some remarks about the use of monads in the implementation of EVADE, and about its exception mechanism.

10.2 The monads for the various components

In the implementation of EVADE, a total of four distinct monads are used. These monads are built from various base monads and monad transformers introduced in the previous chapter. Each of these monads is named after a component that is implemented by it.

The parser monad

The parser is used in EVADE in three places. In the preprocessor, the parser is used to parse evolving algebra specifications. In the run analyzer the parser is used in one place to parse user input to commands, and in another place to parse user input to expressions. In all these places we will use the same parser monad `PM`:

```
type PM  =  ParserT List
```

Thus, parsers of the type `PM a` are list based parsers, without any additional monadic features.

The engine monad

The engine evaluates terms and conditions and performs updates. During these operations, exceptions may be raised. To make this possible, the engine monad `EM` is an exception monad. It is defined as follows:

```
type EM  =  Expt EvadeException
```

The exception type for this monad, called `EvadeException`, will be defined and explained in section 10.3.

The descender monad

The random descender needs to perform monadic random number generation. In order to make this possible, the descender monad `DM` is a state monad specialized in random number generation.

```
type DM  =  State RandomInt
```

The state type, called `RandomInt` represents the seed of the monadic random number generator. The random number generator is explained in section 10.4.5

The explorer monad

The monad of the interactive explorer is called `XM`. The explorer must be able to perform history management, generate random numbers, raise and handle exceptions, and perform interactive I/O. To accommodate these features, the explorer monad must be at once a state monad for history management, a monad for random number generation, an exception monad, and an I/O monad. It is defined as follows:

```
type XM v  =  StateT (Stack (Run v)) (
              StateT RandomInt (
              ExptT EvadeException
              IO))
```

This explorer monad is parameterized with respect to a type variable `v`. This type variable represents the supertype of values of an evolving algebra. We will explain this supertype in section 10.4.3. The base monad of the explorer monad is the I/O monad `IO`. This monad is first transformed by the exception monad transformer. The resulting monad is transformed by the state monad transformer for random number generation. Finally, the state monad transformer is applied to add a stack of runs as a state to the monad. This stack constitutes the history of the run analyzer session. In section 10.6.3, we will explain why the items on this stack are of type `Run v`.

When two components with different monads cooperate, it is necessary to change from the monad of the one to the monad of the other. In the course of the upcoming sections, we will present a number of functions with which these monad-switches can be accomplished in a very elegant manner.

In principle, it is possible to combine all monadic features used by the components of EVADE into a single monad. Each component can than use this monad, even though they do not use all features present in it. The motivation for distributing the monad features over specialized monads is *efficiency*.

Each monad transformer adds an encapsulation to a monad. As the number of transformers increases, both the encapsulated value and the added monad feature become enveloped inside more layers. The later a monad transformer is added, the deeper its corresponding feature will be encapsulated. To access these values and monad features, layers need to be stripped away and re-installed constantly. As a consequence, the evaluation of the monadic program that uses the monad becomes more laborious, and slower.

The most labor intensive subtasks of EVADE are performed by the parser and the engine. To guarantee a good response time of the entire program, especially these components need to run efficiently. However, the monad features used by these components — monadic parsing and exception handling — can impossibly be made outer

layers of a monad in which all necessary monad features are combined. Remember that the I/O monad transformer `IOT` is not suited for interactive I/O (see section 9.6). Hence, we need to use `IO` as the base monad of our all-encompassing monad. As a consequence, the outer layer of this monad will necessarily be provided by the `IO` monad, and the monadic parser and the exceptions will be encapsulated within it. So, since the parser and the engine do not perform I/O, their operation will be hindered constantly by a layer of which they make no use. Needless to say, this situation should be avoided.

The system of four distinct monads in the implementation of EVADE has been designed to avoid unnecessary layers hampering the parser and the engine. The parser monad and the engine monad do not contain any features not used by the parser and the engine, respectively. Only after each of these monads has completed its task, are other monad features added with `lift` and `insert`. Also, the parser feature, which is of no use to the other components is discarded with an `unlift` when parsing is done. The various `lift`, `unlift` and `insert` invocations are grouped inside the monad switching functions that will be presented in the course of the upcoming sections.

10.3 The exception mechanism of EVADE

EVADE is equipped with an extensive exception mechanism. The following exceptions are currently supported:

- Parse exception. Is raised when parsing fails.
- Get, put, and remove exceptions. The first is raised when attempts are made to get the value of a non-existing identifier from a state. The second is raised when an attempt is made to update a non-existing identifier. The last is raised when an attempt is made to remove a name from a state with empty vocabulary.
- Type Exception. Is raised when a type mismatch occurs.
- History exception. Is raised when an attempt is made to recall elements from an empty history stack.

These various exceptions are represented by the following datatype:

```
data EvadeException = GetExp Name | PutExp Name | RemoveExp
                    | TypeExp | HistoryExp | ParseExp
```

The first two exceptions, the get exception and the put exception, have as argument the evolving algebra names whose non-existence caused the exception. To raise these exceptions, we will use the following functions:

- `getExpt name`
- `putExpt name`
- `removeExpt`
- `typeExpt`
- `historyExpt`
- `parseExpt`

How these exception raising functions are defined will be explained in the course of the following sections. To handle these exceptions, two different exception handling functions are available. The first of these handles an exception by producing a run-time error. It is defined as follows:

```
errorHandler e
  = (case e of
       GetExp n    -> error ("Get exception. Unknown name: " ++ n)
       PutExp n    -> error ("Put exception. Unknown name: " ++ n)
       RemoveExp   -> error "Remove exception"
       TypeExp     -> error "Type exception"
       HistoryExp  -> error "History exception"
       ParseExp    -> error "Parse exception")
```

The second exception handler is similar to the first, but returns an output monad in stead of producing a run-time error. It is defined as follows:

```
outputHandler :: (IOMonad m) => EvadeException -> m ()
outputHandler e
  = output "\BEL" >>       -- beep
    (case e of
       GetExp n    -> output ("Get exception. Unknown name: " ++ n)
       PutExp n    -> output ("Put exception. Unknown name: " ++ n)
       RemoveExp   -> output "Remove exception"
       TypeExp     -> output "Type exception"
       HistoryExp  -> output "I can not go back any further"
       ParseExp    -> output "Parse exception")
```

We will use these different handlers for different situations. In programs that perform I/O, the second handler can be used. Each time an exception occurs, a message will be displayed, and the program will continue operating. In programs that do not perform I/O, i.e. programs that produce values of types other than `Dialogue`, the second exception handler can not be used. In these programs we will use the first handler, which produces a run-time error in case of exceptions. In particular, the error exception handler will be used for the compiler, while the output exception handler will be used by the run analyzer.

10.4 Implementation of the low level components

In section 10.1.5 six basic building blocks of EVADE were isolated: the parser, the generator, the evaluator, the executor, the random number generator, and the history manager. We will discuss each of these subcomponents in turn.

10.4.1 Parser

The parser of EVADE is constructed with the monadic parsing library explained in section 9.5. The same method is applied as in the example given there. We start from a BNF syntax description of evolving algebra specifications to define the type of a parse tree. Then, we define a parser for each datatype used. We use the operator `|||=` to remove left-recursiveness.

A complete BNF syntax description of evolving algebra specifications can be found in appendix B. We will not discuss this BNF in full detail. We will focus on the lines of the BNF which define evolving algebra specifications and terms, respectively:

specification → **MODULE** name parameterlist signature
start program stop returnexpr

term → name [(term (, term)*)] | (term) | - term |
number | term termoperator term

These syntax descriptions give rise to the following datatypes:

```
data EASpec =  EASpec Name Parameters Signature
                      StartClause ProgramClause
                      StopClause ReturnClause
data Term   =  ST  Name [Term] |
               UCT UnOp Term |
               BCT Term BinOp Term |
               Num Number
```

Together with the datatypes corresponding to the remaining lines of the syntax description, these datatypes define the type of parse trees of evolving algebras.

For each datatype constitutive of the parse tree type, a parser needs to be defined. To be able to overload the name `parser` for each of these parsers, we introduce the following class of parsers:

```
class ParserClass a where
  parser      :: PM a
```

The member function `parser` of this class yields a parse tree encapsulated by the parser monad as its value. By declaring an instance of this class for each datatype constitutive of the parse tree, a parser is defined for each line in the BNF.

A parser for a complete evolving algebra specification is defined by declaring an instance of `ParserClass` for the datatype `EAspec`:

```
instance ParserClass EASpec where
  parser = spkey "MODULE"  >>
           sp parser       >>= \modulename ->
           sp parser       >>= \parameters ->
           sp parser       >>= \returnexp ->
           sp parser       >>= \signature ->
           sp parser       >>= \start ->
           sp parser       >>= \program ->
           sp parser       >>= \stop ->
           return (EASpec modulename parameters signature
                          start program stop returnexp)
```

This definition corresponds directly to the first line of the syntax description of evolving algebra specifications. It operates by applying the parsers for the clauses of an evolving algebra specification in sequence. Each of these parsers is invoked by an instantation of the overloaded function `parser`.

A parser for evolving algebra terms is defined by declaring an instance of `ParserClass` for the datatype `Term`:

```
instance ParserClass Term where
  parser = ((sp parser             >>= \name ->
             optional (parenthesized (interleaved
                (sp parser) (sptok ",")))  >>= \terms ->
             return (ST name (maybe2list terms)))
            |||
            (parenthesized (sp parser))
            |||
            (sptok "-"              >>
             sp parser              >>= \term ->
             return (UCT Neg term))
            |||
            (number >>= \n ->
             return (Num n)))
           |||= \p ->
           (sp p                    >>= \term1 ->
            sp parser               >>= \op ->
            sp parser               >>= \term2 ->
            return (BCT term1 op term2))

maybe2list Nothing  = []
maybe2list (Just a) = a
```

This parser corresponds directly to the syntax description of terms. It is also very similar to the parser for expressions explaned in section 9.5. For each of the alternatives in the syntax description a sub-parser is defined. These sub-parsers are combined with the monadic plus operator `|||`. To avoid non-termination due to left-recursiveness, the last sub-parser is combined with the others with the operator `|||=`, which is explained in section 9.5 as well.

In this fashion, a parser can be defined for each line in the syntax description of evolving algebra specifications. Together, these parsers form the parser-component of EVADE.

10.4.2 Generator

The task of the generator is to create a Gofer representation of the evolving algebra, on the basis of the parse tree supplied by the parser. Such a Gofer representation has four parts:

- The declaration of the supertype of values.
- The type declaration of the evolving algebra.
- The definition of the evolving algebra.
- The definition of the compiled evolving algebra.

We will discuss these four parts in turn, and explain how they are created by the generator.

Declaration of value supertype

Each name in an evolving algebra takes on values of a particular sort. These sorts are bound to Gofer types by the signature. In order to give a type declaration and a definition of the evolving algebra, these several value types must be united in one supertype. As will be explained in section 10.4.3, such a supertype can be represented in Gofer by a type of the following form:

```
OR A (OR B (OR C .. () ..))
```

where `A`, `B` and `C` are the value types to be united.

The generator deduces which value types occur in an evolving algebra from its signature, and creates the supertype of values which unites them. This supertype is stored in the Gofer representation of the evolving algebra under the name `Typename`, where `name` is the name of the evolving algebra.

Type declaration of the evolving algebra

Apart from the supertype of values, two elements are needed to declare the type of an evolving algebra: the types of its arguments, and the type of its return value. The generator deduces the type of the arguments by looking up in the signature to which Gofer type each argument sort occuring in the parameter list is bound. The result type is deduced by looking up to which Gofer type the result sort is bound.

Suppose the generator deduces that the argument and result type of an evolving algebra are `A1`, `A2` and `R`. Then, the following type declaration of the evolving algebra is produced:

```
nameEA :: A1 -> A2 ->  EA Typename R
```

where `name` is the name of the evolving algebra. The type constructor `EA` is explained in the following subsection.

Definition of the evolving algebra

The type declaration of an evolving algebra involves the type constructor `EA`, which is defined as follows:

```
data EA v r  =  EA Name (EAstate v) Updates Program Formula Term
```

The type variables `v` and `r` represent the value supertype and the result type of the evolving algebra, respectively. The arguments of the constructor function `EA` on the right hand side, correspond to the constituent parts of an evolving algebra definition. We will briefly explain them in turn:

- `Name` The name of the evolving algebra.
- `(EAstate v)` The null-state of the evolving algebra. The type constructor `EAstate` will be explained in section 10.4.3. The notion of a null-state will be explained below.
- `Updates` The parsed start update set of the evolving algebra.
- `Program` The parsed program of the evolving algebra.
- `Formula` The parsed stop condition of the evolving algebra.

- `Term` The parsed return expression of the evolving algebra.

The name of the evolving algebra, its start update set, program, stop condition and return expression need no clarification. These elements are all explicitly present in the evolving algebra specification. Their parsed versions are subtrees of the parse tree produced by the parser. The null-state, however, needs explanation.

The initial state description of an evolving algebra in EVADE has two parts. In the signature, a mapping is given from static sort names and static function names to Gofer types and Gofer functions. This mapping is supplemented by the start update set, which assigns values to dynamic names. The mapping given in the signature defines a so-called null-state, in which all static names have an interpretation, and all dynamic names are undefined. The start update set is fired at this null-state to produce the initial state. The generator constructs the null-state on the basis of the signature.

To construct a definition of an evolving algebra, the generator combines the null-state with the name, start update set, program, stop condition and return expression, which it takes directly from the parse tree.

Definition of the compiled evolving algebra

The last part of the Gofer representation of an evolving algebra is the definition of the compiled evolving algebra. For an evolving algebra with parameters `a1` and `a2`, this definition takes the following form:

```
name a1 a2 = compile (nameEA a1 a2)
```

where `name` is the name of the evolving algebra.

10.4.3 Evaluator

Since the evaluator is part of the engine, it uses the engine monad `EM`. The task of the evaluator is to determine the value of a term or a condition in a given state. Terms and conditions are represented by the types `Term` and `Formula`. In the following subsections, we will explain how values and states are represented, and how the evaluator operates.

Representing values

Each name in an evolving algebra takes on values of a particular sort. These sorts are bound to Gofer types by the signature. Thus, a number of value types is associated to each evolving algebra. These individual value types can be united into a single type, which we will call the value supertype of the evolving algebra. This supertype is needed as the result type of the evaluator and as the type of values to which names are interpreted in an evolving algebra state.

Representing union types

To model union types in Gofer we use the following data constructor:

```
data OR a b  =  L a | R b
```

Using this data constructor, we can define a subtype-supertype relation by the following class and its instances:

```
class SubT sub sup where
  inject     :: sub -> sup
  project    :: sup -> sub

instance SubT a (OR a b) where
  inject         =  L
  project (L x)  =  x
  project _      =  error "Subtype error"

instance (SubT a b) => SubT a (OR c b) where
  inject sub     =  R (inject sub)
  project (R a)  =  project a
  project _      =  error "Subtype error"
```

The class has member function `inject` and `project`. The function `inject` injects a subtype into a supertype. The function `project` projects a supertype to a subtype. The instance declarations together assert that union types are supertypes of the types by which they are formed.

For instance, we can model the union type of integers and booleans as follows:

```
type IntBool  =  OR Int (OR Bool ())
```

To obtain supertype values from subtype values, we use the injection function:

```
inj 1              ==>     L 1
inj True           ==>     R (L True)
```

To obtain subtype values from supertype values, we use projection:

```
prj (L 1)          ==>     1
prj (R (L True))   ==>     True
```

The member functions of the class `SubT` produce run-time errors when type mismatches occur. A monadic counterpart of this class can be defined whose member functions do not produce run-time errors, but raise exceptions. We will call this class `SubType`. The declarations of the class as well as its instances, and an auxilliary class are as follows:

```
class (SubTypeExpt m) => SubType m sub sup where
  inj :: sub -> m sup
  prj :: sup -> m sub

instance (SubTypeExpt m) => SubType m a (OR a b) where
  inj        =  unit . L
  prj (L x)  =  unit x
  prj _      =  subTypeExpt

instance (SubTypeExpt m, SubType m a b) => SubType m a (OR c b) where
  inj sub    =  inj sub  >>= \x ->
                unit (R x)
  prj (R a)  =  prj a
  prj _      =  subTypeExpt
```

```
class (Monad m) => SubTypeExpt m where
  subTypeExpt  :: m a
```

The auxilliary class `SubTypeExpt` has as member function the exception raising function `subTypeExpt`, which was already mentioned in section 10.3. This function is used by the injection and projection functions `inj` and `prj` when a type mismatch occurs.

Representing the value supertype

To model the type of all possible values of an evolving algebra, we start by gathering the non-predefined static sorts of the evolving algebra in a supertype. Assume these sorts to be `S1` and `S2`:

```
type V = OR S1 (OR S2 ())
```

Then, we add the predefined types integer and boolean, as well as the type of dynamic sorts and the type of dynamic sort elements:

```
type Value v        =  OR DS (OR [DS] (OR Bool (OR Int v)))
```

Next, we add the type of functions from values to values.

```
type Fun v          =  [v] -> Opt v
type FunOrVal v     =  OR (Fun v) v
type FunValue v     =  FunOrVal (Value v)
```

We will also use the following abbreviations:

```
type OptValue v     =  Opt (Value v)
type OptFunValue v  =  Opt (FunOrVal (Value v))
```

When `v` is the union of the sorts bound to non-predefined sort names, then `OptValue v` represents the uniontype of values of ground terms, and `OptFunValue` represents the uniontype of values of ground and non-ground terms.

Representing states

Using the type constructor `OptFunValue`, we can define the type of evolving algebra states as follows:

```
type EAstate v  =  [(Name, OptFunValue v)]
```

Again, the type variable `v` represents the union of all non-predefined static sorts of the evolving algebra concerned.

Several operations are needed to inspect and manipulate states. These operations will be given by member functions of a class `StateClass`. The declarations of this class and two auxilliary classes are as follows:

```
class (PreStateClass m s) => StateClass m s v where
  get     :: String ->      s -> m v
  put     :: String -> v -> s -> m s
  extend  :: String -> v -> s -> m s

class PreStateClass m s where
  remove      :: s -> m s
```

```
class StateExpt m where
  removeExpt  ::  m a
  getExpt     ::  Name -> m a
  putExpt     ::  Name -> m a
```

Like the class SubType, the class StateClass contains monadic member functions, which raise expressions when anomalies occur, in stead of producing run-time errors. The relevant exception raising functions are defined in the auxiliary class StateExpt. These functions were mentioned earlier in section 10.3. The member function get is used to obtain the interpretation of a given name in a state. The member function put is used change the interpretation of a given name in a state to a new value. The member function extend adds a new name to a state, and provides it with an initial interpretation. The member function remove undoes the effect of the previous invocation of extend. This last member function can not be defined as a member function of StateClass itself, because Gofer requires the types of member functions to contain all type parameters of the class to which it belongs. Since the type of remove does not involve the type variable v, it does not meet this requirement. For this reason, a superclass PreStateClass which has one type parameter fewer is created to contain remove.

To make the state operations defined in StateClass and PreStateClass available on evolving algebra states, we need to define instances for EAstate of these classes and the auxilliary class StateExpt.

```
instance (Monad m, StateExpt m) => PreStateClass m (EAstate v) where
  remove []     = removeExpt
  remove (x:xs) = unit xs

instance (StateExpt m) => StateClass m (EAstate v) (OptFunValue v) where
  get n []            =  getExpt n
  get n ((m,w):s)     =  if n==m
                         then unit w
                         else get n s
  put n v []          =  putExpt n
  put n v ((m,w):s)   =  if n==m
                         then unit ((m,v):s)
                         else put n v s    >>= \s' ->
                              unit ((m,w):s')
  extend n v s        =  unit ((n,v):s)

instance (ExptMonad EvadeException m) => StateExpt m where
  removeExpt  = raise RemoveExp
  getExpt n   = raise (GetExp n)
  putExpt n   = raise (PutExp n)
```

The get and put operations recursively search a state until the given name is found. If the name is not found, an exception is raised.

Evaluation

The task of the evaluator is to determine the values of a term or a condition in a state. This task is performed by two functions of the following type:

```
evaluateForm :: (EvaluationContext m v)
                => Formula -> EAstate v -> m (OptValue v)
evaluateTerm :: (EvaluationContext m v)
                => Term -> EAstate v -> m (OptValue v)
```

These functions are fairly straightforward translations of the evaluation guidelines for terms and conditions given in section `EvalGuidelines`. Take for example the guideline for evaluating a term consisting of a 0-ary function name:

- The value of a term f in state S is the interpretation of the 0-ary function identifier f in S. If the interpretation of the function identifier is **sf** Undef, so is the value of the term.

Hence, to determine the value of the 0-ary function name f in S, we must first get the interpretation of f from S. Next, we must test whether the interpretation is **sf** Undef or not. If it is, the value to be returned is **sf** Undef as well. If it is not, the value to be returned is the interpretation of f itself. Hence, the function `evaluateTerm` is defined for 0-ary function names as follows:

```
evaluateTerm (ST n []) s
  =  get n s  >>= \ofv ->
     case ofv of
       Undef   -> unit Undef
       Def fv  -> prjVal fv    >>= \v ->
                  unit (Def v)
```

The variables `ofv`, `fv` and `v` are acronyms of their types: `OptFunValue v`, `FunValue v`, and `Value v`. The function `prjVal` projects a value of type `FunValue v` to a value of type `Value v`, which represents the type of ground terms.

The other guidelines for term evaluation are translated in a similar manner to function bindings of `evaluateTerm`. These function bindings have been supplemented with definitions for evaluation of the predefined arithmetic functions.

10.4.4 Executor

Like the evaluator, the executor is part of the engine and consequently uses the engine monad `EM`. The task of the executor is to perform updates upon a state. In section 2.6 the effect of firing a set of updates in parallel at a state S was defined as the accumulation of the effects of the individual updates. The accumulation of these individual effects proceeds by firing the updates sequentially. Each update is fired at the state produced by its predecessor, but all updates are evaluated in S.

The function `executeU s s0` evaluates an update in state `s0`, and fires it at state `s`. It is defined as follows:

```
executeU    :: (EvaluationContext m v)
               => Update -> EAstate v -> EAstate v -> m (EAstate v)
executeU (SU u) s s0  =  executeSU u s s0
executeU (NU u) s s0  =  executeNU u s s0
```

The auxiliary function `executeSU` and `executeNU` are used to perform a local function update and a new-update, respectively. These functions are defined in accordance with the description of the effects of local function updates and new-updates in section 2.6.

They make use of the function `evaluateTerm` to determine the values of right hand sides, and other terms.

Using the function `executeU`, the function `executeUsSeq s s0` can be defined. This latter function fires an update set sequentially at the state `s`, while each update is evaluated in `s0`. It is defined as follows:

```
executeUsSeq    :: (EvaluationContext m v)
                   => Updates -> EAstate v -> EAstate v -> m (EAstate v)
executeUsSeq []      s s0  =  unit s
executeUsSeq (u:us) s s0  =  executeU u s s0 >>= \s' ->
                              executeUsSeq us s' s0
```

This function definition accords with the description of the accumulation of effects of updates in a set.

10.4.5 Random number generator

The monadic random number generator is defined in terms of a state monad. The state of this state monad represents the current seed of the generator. Its type is `RandomInt`. Three operations are defined for the random number generator:

- `reseed n`: reseeds the generator with `n`.
- `generate`: generates a random number.
- `random n`: generates a random number between `0` and `n-1`.

These functions are defined in terms of the state monad function `update`, and the function `minimumStandardGenerator`, which is the actual implementation of the random number generator. Here we only list the type declarations of the monadic number generator functions:

```
reseed   :: (StateMonad RandomInt m) => Int -> m ()
generate :: (StateMonad RandomInt m) => m Int
random   :: (StateMonad RandomInt m) => Int -> m Int
```

These functions can be used in any monad `m` that satisfies the type predicate `StateMonad RandomInt m`.

10.4.6 History manager

The task of the history manager is to enable states of the run analyzer to be archived and recalled. The history manager is based on a state monad and an exception monad. Three functions are defined for history management.

- `initarchive` Initialize the archive.
- `archive` Archive an item.
- `recall` Recall the most recently archived item.

These functions, as well as an auxiliary class are defined as follows:

```
initarchive :: (StateMonad (Stack a) m, HistException m)
               => m (Stack a)
initarchive  =  update (const (Stack []))

archive :: (StateMonad (Stack a) m, HistException m)
           => a -> m ()
archive a  =  update (push a) >>
              unit ()

recall  :: (StateMonad (Stack a) m, HistException m)
           => m a
recall = update id >>= \stack ->
         case poptop stack of
           (stack',Nothing) -> historyExpt
           (stack',Just x)  -> update (const stack') >>
                               unit x

class HistException m where
  historyExpt  :: m a
```

The auxiliary class `HistException` harbors the exception raising function `historyExpt`. This function is used by `recall` if no items are present in the archive. These three functions for history management are defined in terms of the stack functions `push` and `poptop`.

10.5 Implementation of the intermediate level components

In the foregoing section, the six low level components of EVADE were discussed. We turn now to the intermediate level components: the engine, the random descender, and the interactive explorer.

10.5.1 Engine

The engine of EVADE generates the transition tree which is induced by an evolving algebra. To represent transition trees, a type of general trees called `GeneralTree` is used. On trees of this type, several functions are defined:

- `root` Returns the root node of a tree.
- `branches` Returns the list of branches of a tree.
- `consTrees` Constructs a tree from a root and a list of branches.

The transition tree of an evolving algebra, which can be viewed as a branching run, can be represented as follows:

```
type Run v  =  GeneralTree (Name, EAstate v)
```

The nodes of these transition trees are pairs of transition rule names and evolving algebra states. The transition rule name indicates which rule was fired to obtain the state.

Using the type constructor `Run`, we can define the engine as follows:

```
engine :: (EvaluationContext m v)
          => EA v r -> EM (Run v)
engine ea
  =  let EA _ s startUs program _ = ea
     in executeUsSeq startUs s s >>= \s0 ->
        doProg program "START" s0
```

Hence, the engine takes an evolving algebra and returns a run encapsulated in an engine monad. The function `executeUsSeq` is used to fire the start update set at the null-state of the evolving algebra, which results in the initial state `s0`. The auxiliary function `doProg` generates the run induced by the evolving algebra, on the basis of its program. This latter function is defined as follows:

```
doProg          :: (EvaluationContext m v)
                   => Program -> Name -> EAstate v -> EM (Run v)
doProg p n s0
  =  unit (map unEM
                  [ evaluateForm g s0    >>= \ov ->
                    prj (unOpt ov)       >>= \b ->
                    if not b
                    then unit NilTree
                    else executeUsSeq u s0 s0 >>= \s ->
                         doProg p n s  |  (n,g,u) <- p ]) >>= \runs ->
     unit (consTree (n,s0) runs)
```

Note that the bulk of this function definition is formed by a list comprehension. Inside this list comprehension, three things are done for each rule in the program. First, the guard `g` of the rule is evaluated in `s0`. Secondly, the resulting value `ov` is projected to a boolean value `b`. Thirdly, if this boolean value is `True`, then the update set of the rule `u` is fired at state `s0`, and the function `doProg` is invoked recursively with the resulting state `s`. Thus, this list comprehension generates a list of transition subtrees – one subtree for each applicable rule.

The subtrees in the list created by the list comprehension are all encapsulated by engine monads. To free the subtrees from their encapsulation, the function `unEM` is mapped over the list. This function is defined as follows:

```
unEM :: EM a -> a
unEM em
  =  case em of
      Succ a  -> a
      Fail e  -> errorHandler e
```

Exceptions are handled by this function by applying the error handler. If no exception occured, the value `a` is returned unencapsulated.

In the last line of the function definition of `doProg`, the complete transition tree is constructed with the state `s0` as root, and the unencapsulated subtrees as branches.

10.5.2 Random descender

The task of the random descender is to descend the transition tree of an evolving algebra along a random path until a final state is reached. In this final state, the return expression of the evolving algebra must be evaluated. At each fork encountered

in the transition tree, the random descender will use the monadic parser generator to choose between the various branches.

The subcomponent of the descender, the engine, uses the engine monad `EM`. The descender itself uses the descender monad `DM`, which is distinct from the engine monad. Hence, to be able to use the engine, the descender needs a function for switching from engine monads to descender monads. This function is called `fromEMtoDM`, and is defined as follows:

```
fromEMtoDM :: EM a -> DM a
fromEMtoDM em
  =  case handle (errorHandler, em) of
       Succ a  ->  unit a
       Fail a  ->  error "Exception was raised, but not handled."
```

Recall the definitions of the engine monad and the descender monad:

```
type EM   =  Expt EvadeException
type DM   =  State RandomInt
```

The function `fromEMtoDM` applies the error handler to the engine monad. If any exceptions slip through this handler, a run-time error is generated. Otherwise, the value `a` encapsulated by the engine monad is inserted into the descender monad by `unit`.

Using the function `fromEMtoDM`, the descender and an auxiliary function are defined as follows:

```
descend :: (EvaluationContext EM t,
            CompilerContext t,
            SubType EM r (Value t))
        =>  Term -> Run t -> DM r
descend retexpr run
  =  reseed descenderSeed        >>
     descend' run                >>= \s' ->
     fromEMtoDM (
        evaluateTerm retexpr s' >>= \ov ->
     case ov of
       Def v -> prj v
       Undef -> error "Return expression of evolving algebra is <undef>.")

descend' :: (EvaluationContext EM t,
             CompilerContext t)
         =>  Run t -> DM (EAstate t)
descend' run
  =  case (branches run) of
       []  -> unit (snd (root run))
       [s] -> descend' s
       ss  -> random (length ss) >>= \n ->
              descend' (ss!!n)
```

In the first line of the definition of the function `descend`, the random generator is initialized with a seed. The auxiliary function `descend'` recursively descends the transition tree, and returns a final state. In this final state, the return expression of

the evolving algebra is evaluated by the evaluator. The function `fromEMtoDM` switches from engine monad to descender monad. If the return value is undefined, a run-time error is produced. Otherwise, the return value is projected from the value supertype to the return type of the evolving algebra.

10.5.3 Interactive explorer

The general setup of the interactive explorer is as follows. The explorer iterates in a loop of parsing a command and executing a command. If during the execution of a command, an exception is raised, the output handler is applied and the loop is started anew.

```
explore_loop    :: (ExploreContext t)
                   => Run t -> XM t ()
explore_loop r
  =  newline                          >>
     prompt commandPrompt             >>= \i ->
     handle (exploreHandler r,
     fromPMtoXM (goodparse i parser)  >>= \c ->
     execute c r)
```

To switch from the parser monad `PM` to the explorer monad `XM`, the function `fromPMtoXM` is used. This function is defined as follows:

```
fromPMtoXM    :: PM a -> XM t a
fromPMtoXM    =  fromEMtoXM.fromPMtoEM

fromEMtoXM    :: EM a -> XM t a
fromEMtoXM m  =  lift (lift (insert m))
```

Thus, the function `fromPmtoXM` first applies the function `fromPmtoEM` to switch from the parser monad to the engine monad. Then, the function `fromEMtoXM` is applied to switch further to the explorer monad. This latter function is defined in terms of two lifts, and one insertion. Recall the definitions of the engine monad and the explorer monad:

```
type EM    =  Expt EvadeException
type XM t  =  StateT (Stack (Run t)) (
              StateT RandomInt (
              ExptT EvadeException
              IO))
```

The insert operation performed on the engine monad inserts the monad `IO` into the exception monad, to produce an exception monad transformer. The two succesive lifts turn the result into the explorer monad by adding two state monad tranformers.

The function `execute` executes a parsed command. Six distinct commands can be distinguished: quit, refresh, eval, back, step, and until. The first three commands serve to inspect the current evolving algebra state, and to exit the run analyzer. The last three commands serve to navigate through the transition tree.

Executution of the refresh command

When the the user issues the refresh command, the current evolving algebra state is shown. The execution of the refresh command is defined as follows:

```
execute Refresh r
  =  showState (root r) >>
     explore_loop r
```

After showing the current state, the explorer loop is continued.

Executution of the quit command

When the user issues the quit command, the interactive explorer is exit-ed. The execution of the quit command is defined as follows:

```
execute Quit r
  =  nop
```

Thus, no action is undertaken, and the explorer loop is not continued.

Executution of the eval command

When the user issues the eval command, he is prompted to type an expression. This expression is parsed and evaluated, and the value is displayed.

```
execute Eval r
  =  prompt "Expression >> "                   >>= \i ->
     fromEMtoXM (fromPMtoEM (goodparse i parser) >>= \f  ->
                   evaluateForm f (snd (root r)))  >>= \ov ->
     output "Value: "                          >>
     output (showval ov)                       >>
     explore_loop r
```

The functions `fromPMtoEM` and `fromEMtoXM` are used to switch from the parser monad to the enigine monad and from the engine monad to the explorer monad, respectively. After the value of the expression has been displayed, the explorer loop is continued.

The three commands for navigation through the transition tree are the back, step and until commands. At each moment during the navigation one of the evolving algebra states in the transition tree of an evolving algebra is the current state. The navigation commands operate by making a different state current. To make backtracking possible, the history manager is used. When an evolving algebra state ceases to be the current state, the subtree of which it is the root is stored on the stack. Hence, the items that are stored on the history stack are not single evolving algebras, but transition sub-trees. When the run analyzer needs to choose beteen several possible successor states, he invokes the monadic random number generator.

Executution of the back command

When the user issues the back command, the run analyzer returns to the state that was previously current. The execution of this command is defined as follows:

```
execute Back r
  =  recall                               >>= \predecessor ->
     output "Return to previous state\n"  >>
     showState (root predecessor)         >>
     explore_loop predecessor
```

In the first line, the previous run analyzer state is recalled from the history stack. After the new current evolving algebra state has been shown, the explorer loop is continued.

Executution of the step command

The step command allows the user to descend a single step down the transition tree. The execution of the step command is defined as follows:

```
execute Step r
  =  step_one r                           >>= \successor ->
     showRuleName (fst (root successor))  >>
     showState (root successor)           >>
     explore_loop successor
```

The auxiliary function `step_one` is used to select a successor of the current state. If several successors exist, this auxiliary function consults the user to choose between them or applies the random number generator. The rule which was applied to obtain the successor is displayed, as well as the successor state itself. Finally, the explorer loop is continued.

Executution of the until command

The until command prompts the user for a stop conditions, and subsequenlty runs the evolving algebra until a state is encountered in which the stop condition is true. The exectution of the until command is defined as follows:

```
execute Until r
  =  prompt "Stop condition >> "          >>= \i ->
     fromPMtoXM (goodparse i parser)      >>= \f ->
     archive r                            >>
     run_until r f                        >>= \successor->
     showState (root successor)           >>
     explore_loop successor
```

The auxiliary function `run_until` is used to descend the transition graph. At forks in the transition graphs, this function always invokes the random number generator. When the a state in which the stop condition is true is encountered, this state is displayed. Finally, the explorer loop continues.

10.6 Implementation of the top level components

The low level and intermediate level subcomponents of EVADE have now been explained. The top level components, the preprocessor, the compiler and the run analyzer, are built from these subcomponents. We will discuss these top level components in turn.

10.6.1 Preprocessor

The task of the preprocessor is to convert an evolving algebra specification to an intermediary representation, which can be read by the compiler and the run analyzer. Evolving algebra specifications are assumed to reside in files with extension `ea`. The preprocessor reads such a specification from file and feeds it to its first subcomponent: the parser. The resulting parse tree is then given to the second subcomponent: the generator. From this parse tree, the generator generates a Gofer representation of the evolving algebra. Finally, this Gofer representation is written to a file with extension `gea`. The definition of the preprocessor and an auxiliary function is as follows:

```
preprocess        :: String -> Dialogue
preprocess fname  =  readFile  (fname++".ea")  abort (\i ->
                     writeFile (fname++".gea") (preprocess' i) abort done)

preprocess'       :: String -> String
preprocess' s     =  generateEA (unPM (parser :: PM EASpec) s)
```

The functions `readFile` and `writeFile` are predefined `Dialogue` functions for reading and writing files in continuation style. The auxiliary function `preprocess'` applies the monadic parser to the input string which originates from the evolving algebra specification file. As a result, a parse tree of an evolving algebra is obtained which is encapsulated by the parser monads `PM`. Before the generator can be applied to the parse tree, the monad must be stripped away. This is done by the function `unPM`, which is defined as follows:

```
unPM :: PM a -> String -> a
unPM p s
  =  case unlift (goodparse s p) of
       []      ->  error "Parse failed"
       (a:as)  ->  a
```

Recall that the parser monad was defined as follows:

```
type PM  =  ParserT List
```

The function `unPM` unlifts this monad to the underlying monad `List`. If the resulting list is empty, a parse failure has occured. If the list is non-empty, the first of the possible parse trees is selected as the result.

10.6.2 Compiler

The task of EVADE's compiler is twofold. First, the compiler must invoke the engine to lazily generate the transition graph of an evolving algebra. Secondly, the random descender must be applied to this transition graph in order to select a final state from this tree in a non-determinate fashion, and to evaluate the return expression in this final state. The two subcomponents of the compiler, the engine and the random descender, use different monads. The compiler switches from the engine monad `EM` to the descender monad `DM` with the function `fromEMtoDM`, which was defined in section 10.5.2. To strip away the descender monad from the final result of the descender, the following function is used:

```
unDM         :: DM a -> a
unDM dm      =  let (a,_) = (unState dm) undefined
                in a
```

Recall that the state monad and its auxiliary function are defined as follows:

```
data State s a = State (s -> (a,s))
unState (State x) = x
```

The function `unDM` operates by stripping away the tag from the state monad with `unState`, and feeding the state transformer with an undefined initial state. The result is a pair `(a,s)` of a value and a state. With pattern matching the value `a` is selected and returned.

The compiler is defined in terms of the function `fromEMtoDM` and `unDM` as follows:

```
compile      :: (EvaluationContext EM t,
                 CompilerContext t,
                 SubType EM r (Value t))
             =>  EA t r -> r
compile ea
  =  let retvalue  =  (fromEMtoDM (engine ea)) >>= \r ->
                       descend retexpr r
         (EA _ _ _ _ retexpr) =  ea
     in unDM retvalue
```

In the first line of the function definition, the engine is applied to the evolving algebra, yielding a transition tree `r`. In the second line, the random descender is called with the return expression of the evolving algebra and its transition tree as arguments. Finally, in the last line, the return value produced by the descender is freed from the descender monad, and returned.

10.6.3 Run analyzer

The task of the run analyzer is twofold. Firstly, it must apply the engine to an evolving algebra to obtain the transition graph induced by it. Secondly, it must apply the interactive explorer to the transition tree. To switch from the engine monad to the explorer monad, it uses the function `fromEMtoXM`. The run analyzer is defined as follows:

```
analyze      :: (EvaluationContext EM t,
                 ExploreContext t,
                 InteractiveIO (XM t))
             => EA t r -> Dialogue
analyze ea
  = dialogue (handle (outputHandler,
                      (fromEMtoXM (engine ea) >>= \r ->
                       explore r)))
```

If any exceptions occur during the operation of the engine or the explorer, the output handler is applied to handle them. The function `dialogue` is a member of the type class `InteractiveIO`. It transforms an I/O monad to `Dialogue`, which is Gofers type for interactive I/O.

10.7 Extendability of EVADE

In this section, we will evaluate the implementation of EVADE with respect to its extendability. Especially the benefit of monads will be asessed. Also, we will sketch possible future implementations of the features that we observed to be missing from EVADE in section 7.3.

In the implementation of EVADE, we have not aimed for maximal performance, but for perspicuity and extendability of the program. To obtain these properties, we have applied several methods. The first of these methods is the commonly adopted device of functional decomposition. In section 10.1 we explained the hierarchy of functional components that constitute EVADE. Within the basic building blocks isolated in this decomposition, even smaller elements can be distinguished. Secondly, we have made ample use of Gofer's various forms of polymorphism to introduce generality into the elements of the implementation where this was appropriate. But our most important means of achieving perspicuity and extendability in the implemention of EVADE, has been the monadic programming method.

Using monads, we have been able to incorporate imperative programming features into a functional program in a very elegant way. Among these features are interactive I/O, an exception mechanism, and states for history management and random number generation. Incorporation of these features into a non-monadic program lead to highly cluttered code. For instance, to implement a state in a non-monadic program generally implies adding an argument to each of its functions, through which the state can be passed around. As a result, local portions of code can no longer be understood without taking the entire program into consideration. Also, the imperative feature itself looses its attractive simplicity. In the monadic implementation of EVADE, on the other hand, imperative features have been incorporated without compromising the perspicuity of the code or the imperative nature of the features.

The monadic programming method has effected extendability as well as perspicuity. In fact, during its development, EVADE has already undergone several extensions to take on its present form. In particular, both the history manager and the random number generator have been added in a later stage. Both these extensions involved two local changes. Firstly, the appropriate monad was enlarged by applying a monad transformer to it. Secondly, the program was locally changed to capitalize on the added monad features. In general, monadic programs can be extended without causing an avalanche of adaptations throughout the existing code.

In section 7.3 we observed that integrity constraints and run-time statistics are not supported by EVADE, but might be added to the program with reasonable little programming effort, and without drastic changes to existing code. We can now substantiate this claim. Both these features can be implemented by the two-step process of enlarging a monad and locally changing code. For both these extensions, the state monad transformer can be used. In the first case a list of evolving algebra conditions, is the appropriate state type. Watching the constraints can be realized by inserting `inspect` and `sentinel` functions at appropriate places into the existing code. In the second case, the approriate state type might be a series of various counters. Statistics can be gathered by inserting invocations of the `update` function. Hence, these extensions leave the existing program unaffected, and do not compromise the perspicuity of the code.

EVADE's monadic parser is also geared to extension. This is both due to the monadic nature of the parser and to the overloading of the function `parser` by the help of the class `ParserClass`. EVADE can be made to accept a new language construct

for evolving algebra specifications by making changes of two kinds. Firstly, a instance of the parser class must be declared to specify a sub-parser for the new construct. Secondly, the new parser must be merged into the existing parser. Due to the monadic structure of the parser, this merging operation will in general require only a local insertion of an invocation of `parser`. In section 7.3, three possible future extensions of EVADE were mentioned, which demand new language constructs: macros, the remove update, and variable declarations for massive parallelism. EVADE's parser can be extended to accept each of these language constructs without changing the existing parser code.

The extension of EVADE with macros, the remove update and massive parallelism requires changes not only to the parser, but also to the engine. We can not assess how drastic these changes will need to be, without subjecting the relevant features to a more thorough investigation. However, a few general observations can be made. The componets of the engine, the executor and the evaluator, can be read as definitions of the semantics of evolving algebra updates and terms. The proposed new features are each intended to *add* to the semantics of evolving algebras, not to *change* it. Hence, their implementation can be expected to require mostly additions to, and no modifications of the existing evaluator and executor.

Appendix A

Monad Proofs

In chapter 9 the monadic programming method was presented. The definitions of monads, monad extensions and monad transformers, given in that chapter, involved a number of laws. Three groups of laws can be distinguished: monad laws, monad extension laws and monad transformer laws. In this chapter, these laws are proven to hold for a number of particular monads, monad extensions and monad transformers.

We will not give any proofs concerning the exception monad, the exception monad transformer or the parser monad transformer. These monadic entities are very similar to the maybe monad, the maybe monad transformer and the state monad transformer respectively. As a consequence, the proof concerning them are almost identical. For this reason, these proofs have been omitted from this appendix.

A.1 Proofs of the monad laws

Three monad laws were presented in section 9.1:

```
Left unit:      (unit a) `bind` k  =  k a
Right unit:     m `bind` unit  =  m
Associativity:  m `bind` (\a -> (k a) `bind` (\b -> h b))
                  =  (m `bind` (\a -> k a)) `bind` (\b -> h b)
```

We will prove these laws to hold for the identity monad, the maybe monad, the list monad, and the state monad.

The `Id` monad

First monad law: left unit

```
(unit x) `bind` k
    =  (Id x) `bind` k        def. unit
    =  k x                    def. bind
```

Second monad law: right unit

```
m `bind` unit
  (Assume: m = Id x)
    =  (Id x) `bind` unit     by assumption
```

```
    =  (Id x) `bind` Id       def. unit
    =  Id x                   def. bind
    =  m                      by assumption
```

Third monad law: associativity

```
m `bind` (\a -> (k a) `bind` (\b -> h b))
  (Assume: m = Id x)
    =  (Id x) `bind` (\a -> (k a) `bind` (\b -> h b))    by assumption
    =  (k x) `bind` (\b -> h b)                          def. bind
    =  ((Id x) `bind` k) `bind` (\b -> h b)              def. bind
    =  ((Id x) `bind` (\a -> k a)) `bind` (\b -> h b)
    =  (m `bind` (\a -> k a)) `bind` (\b -> h b)         by assumption
```

The `Maybe` monad

First monad law: left unit

```
(unit x) `bind` k
    =  (Just x) `bind` k      def. unit
    =  k x                    def. bind
```

Second monad law: right unit

```
m `bind` unit
  (Assume: m = Just x)
    =  (Just x) `bind` unit      by assumption
    =  (Just x) `bind` Just      def. unit
    =  Just x                    def. bind
    =  m                         by assumption
  (Assume: m = Nothing)
    =  Nothing `bind` unit       by assumption
    =  Nothing `bind` Just       def. unit
    =  Nothing                   def. bind
    =  m                         by assumption
```

Third monad law: associativity

```
m `bind` (\a -> (k a) `bind` (\b -> h b))
  (Assume: m = Just x)
    =  (Just x) `bind` (\a -> (k a) `bind` (\b -> h b))
    =  (k x) `bind` (\b -> h b)                          def. bind
    =  ((Just x) `bind` k) `bind` (\b -> h b)            def. bind
    =  ((Just x) `bind` (\a -> k a)) `bind` (\b -> h b)
    =  (m `bind` (\a -> k a)) `bind` (\b -> h b)         by assumption
  (Assume: m = Nothing)
    =  Nothing `bind` (\a -> (k a) `bind` (\b -> h b))
    =  Nothing                                           def. bind
    =  Nothing `bind`(\b -> h b)                         def. bind
    =  (Nothing `bind` (\a -> k a)) `bind`(\b -> h b)    def. bind
    =  (m `bind` (\a -> k a)) `bind` (\b -> h b)         by assumption
```

The `List` monad

First monad law: left unit

```
(unit x) `bind` k
    =  [x] `bind` k                 def. unit
    =  [ b | a <- [x], b <- k a]    def. bind
    =  [ b | b <- k x]
    =  k x
```

Second monad law: right unit

```
m `bind` unit
    =  [ b | a <- m, b <- unit a ]   def. bind
    =  [ b | a <- m, b <- [a] ]      def. unit
    =  [ b | b <- m]
    =  m
```

Third monad law: associativity

```
m `bind` (\a -> (k a) `bind` (\b -> h b))
    =  [ b | a <- m, b <- ((k a) `bind` (\b -> h b))]      def. bind
    =  [ b | a <- m, b <- [ b | a' <- k a, b <- h a']]     def. bind
    =  [ b | a <- m, a' <- k a, b <- h a']
    =  [ b | a' <- [ a' | a <- m, a' <- k a], b <- h a']
    =  [ b | a' <- (m `bind` (\a -> k a)), b <- h a']      def. bind
    =  (m `bind` (\a -> k a)) `bind` (\b -> h b)           def. bind
```

The `State` monad

In order to condense the proofs for the state monad, we have abbreviated `State` and `unState` by `St` and `unSt` in a number of places.

First monad law: left unit

```
(unit x) `bind` k
    =  State (\s -> (s,x)) `bind` k                        def. unit
    =  State (\s -> let (s',x) = unState (State (\s -> (s,x))) s
                    in unState (k x) s')                   def. bind
    =  State (\s -> let (s',x) = (\s -> (s,x)) s
                    in unState (k x) s')                   def. unState
    =  State (\s -> let (s',x) = (s,x)
                    in unState (k x) s')
    =  State (\s -> unState (k x) s)
  (Assume: k x = State (\s -> k' (s,x)))
    =  State (\s -> unState (State (\s -> k' (s,x))) s)
    =  State (\s -> k' (s,x))                              def. unState
    =  k x                                                 assumption
```

Second monad law: right unit

```
m `bind` unit
    =  State (\s -> let (s',a) = unState m s
```

```
                 in unState (unit a) s')             def. bind
   =  State (\s -> let (s',a) = unState m s
                 in unState (State (\s -> (s,a))) s')   def. unit
   =  State (\s -> let (s',a) = unState m s
                 in (\s -> (s,a)) s')                def. unState
   =  State (\s -> let (s',a) = unState m s
                 in (s',a))
   =  State (\s -> unState m s)
 (Assume: m = State m')
   =  State (\s -> unState (State m') s)
   =  State (\s -> m' s)                             def. unState
   =  State m'
   =  m                                              assumption
```

Third monad law: associativity

```
m `bind` (\a -> (k a) `bind` (\b -> h b))
   =  St (\s -> let (s',a) = unSt m s
                in unSt ((k a) `bind` (\b -> h b)) s')        def. bind
   =  St (\s -> let (s',a) = unSt m s
                in unSt (St (\s -> let (s'',a) = unSt (k a) s
                                   in unSt (h a) s'')) s')     def. unit
   =  St (\s -> let (s',a) = unSt m s
                in (\s -> let (s'',a) = unSt (k a) s
                          in unSt (h a) s'') s')              def. unSt
   =  St (\s -> let (s',a) = unSt m s
                in let (s'',a) = unSt (k a) s'
                   in unSt (h a) s'')
   =  St (\s -> let (s'',a) = let (s',a) = unSt m s
                              in unSt (k a) s'
                in unSt (h a) s'')
   =  St (\s -> let (s'',a) = unSt (St (\s -> let (s',a) = unSt m s
                                              in unSt (k a) s')) s
                in unSt (h a) s'')                            def. unSt
   =  St (\s -> let (s'',a) = unSt (St (m `bind` (\a -> k a))) s
                in unSt (h a) s'')                            def. unit
   =  (m `bind` (\a -> k a)) `bind` (\b -> h b)               def. bind
```

A.2 Proofs of the monad extension laws

In section 9.2, laws were presented for two monad extensions: the zero extension and the plus extension. Hence, two groups of monad extension laws can be distinguished. The zero laws are:

```
Left zero:   zero `bind` k = zero
Right zero:  m `bind` (\a -> zero)  =  zero
```

The plus laws are:

```
zero ||| m     =  m
m    ||| zero  =  m
(a = zero  /\  b = zero) iff (a ||| b  =  zero)
```

The zero extension and the plus extension were only defined for the maybe monad and the list monad. Hence, we need to prove the monad extension laws only for these two monads.

The `Maybe` monad

First zero law: left zero

```
zero `bind` k
    =  Nothing `bind` k         def. zero
    =  Nothing                  def. bind
    =  zero                     def. zero
```

Second zero law: right zero

```
m `bind` \a -> zero
  (Assume: m = Just x)
    =  (Just x) `bind` \a -> zero       by assumption
    =  (\a -> zero) x                   def. bind
    =  zero
```

First plus law

```
zero ||| m
    =  Nothing ||| m          def. zero
    =  m                      def. |||
```

Second plus law

```
m ||| zero
    =  m ||| Nothing          def. zero
  (Assume: m = Just x)
    =  (Just x) ||| Nothing
    =  Just x                 def. |||
    =  m                      by assumption
  (Assume: m = Nothing)
    =  Nothing ||| Nothing
    =  Nothing                def. |||
    =  m                      by assumption
```

Third plus law

The third plus law is a double implication:

```
(a = zero  /\  b = zero) iff (a ||| b  =  zero)
```

We will prove the two implications separately. The first implication is proven as follows:

```
a = zero  /\  b = zero
  =>  a ||| b
        =  zero ||| zero
        =  zero              The first plus law
```

The second implication can be proven by non-contradiction and analysis of cases. Suppose the consequence `a = zero /\b = zero` is false. We can then distinguish three cases: `a /= zero /\b = zero` or `a = zero /\b /= zero` or `a /= zero /\b /= zero`. In all these cases we obtain contradictions as follows:

```
a ||| b = zero /\ a /= zero /\ b = zero
  =>  a ||| b
        =  a ||| zero    b = zero
        =  a             second plus law
        /=  zero         a /= zero

a ||| b = zero /\ a = zero /\ b /= zero
  =>  a ||| b
        =  zero ||| b    a = zero
        =  b             first plus law
        /=  zero         b /= zero

a ||| b = zero /\ a /= zero /\ b /= zero
  =>  a ||| b
        (Assume: a = Just x)
          =  Just x ||| b         def. |||
          =  Just x               def. |||
          =  a                    assumption
          /=  zero                a /= zero
```

Since a contradiction occurs in all cases, the consequence must be true when the antecedent is. Hence, the second implication holds.

The `List` monad

First zero law: left zero

```
zero `bind` k
    =  [] `bind` k                      def. zero
    =  [ b | a <- [], b <- k a]         def. bind
    =  []
    =  zero                             def. zero
```

Second zero law: right zero

```
m `bind` \a -> zero
    =  m `bind` \a -> []                def. zero
    =  [ b | a <- m, b <- []]           def. bind
    =  []
    =  zero                             def. zero
```

First plus law

```
zero ||| m
    =  [] ||| m            def. zero
    =  [] ++ m
    =  m
```

Second plus law

```
m ||| zero
    =  m ||| []           def. zero
    =  m ++ []
    =  m
```

Third plus law

We will prove the third law again by proving its two constituent implications separately. The first implication is proven as follows:

```
a = zero  /\  b = zero
  =>  a ||| b
        =  zero ||| zero
        =  zero              The first plus law
```

The second implication can be proven by non-contradiction and analysis of cases. Suppose the consequence `a = zero /\b = zero` is false. We can then distinguish three cases: `a /= zero /\b = zero` or `a = zero /\b /= zero` or `a /= zero /\b /= zero`. In all these cases we obtain contradictions as follows:

```
a ||| b = zero /\ a /= zero /\ b = zero
  =>  a ||| b
        =  a ||| zero    b = zero
        =  a             second plus law
        /=  zero         a /= zero

a ||| b = zero /\ a = zero /\ b /= zero
  =>  a ||| b
        =  zero ||| b    a = zero
        =  b             first plus law
        /=  zero         b /= zero

a ||| b = zero /\ a /= zero /\ b /= zero
  =>  a ||| b
        (Assume: a = Just x)
          =  (x:xs) ||| b
          =  (x:xs) ++ b       def. |||
          /=  zero
```

Since a contradiction occurs in all cases, the consequence must be true when the antecedent is. Hence, the second implication holds.

A.3 Proofs of the monad transformer laws

In section 9.3, the following monad laws were presented:

```
Unit lift:  lift (unit a)  =  unit a
Bind lift:  lift (m `bind` \a -> k a)
              =  (lift m) `bind` (\a -> lift (k a))
Unlift:     unlift (lift m)  =  m
```

We will present proofs for these laws for the maybe monad transformer and the state monad transformer.

The `MaybeT` monad transformer

First monad transformer law

```
lift (unit a)
    =  MaybeM (unit a `bind` \a ->
               unit (Just a))
    =  MaybeM (unit (Just a))
    =  unit a
```

Second monad transformer law

```
(lift m) `bind` (\a -> lift (k a))
    =  MaybeM (unMaybeM (lift m) `bind` \maybea ->
               case maybea of
                 Just a  -> unMaybeM (lift (k a))
                 Nothing -> unit Nothing)
    =  MaybeM (unMaybeM (MaybeM (m `bind` \a ->
                                 unit (Just a))) `bind` \maybea ->
               case maybea of
                 Just a  -> unMaybeM (lift (k a))
                 Nothing -> unit Nothing)
    =  MaybeM (m `bind` \a ->
               unit (Just a) `bind` \maybea ->
               case maybea of
                 Just a  -> unMaybeM (lift (k a))
                 Nothing -> unit Nothing)
    =  MaybeM (m `bind` \a ->
               case (Just a) of
                 Just a  -> unMaybeM (lift (k a))
                 Nothing -> unit Nothing)
    =  MaybeM (m `bind` \a ->
               unMaybeM (lift (k a)))
    =  MaybeM (m `bind` \a ->
               unMaybeM (MaybeM ((k a) `bind` \a ->
                                  unit (Just a))))
    =  MaybeM (m `bind` \a ->
               (k a) `bind` \a ->
               unit (Just a))
    =  MaybeM ((m `bind` \a ->
                k a) `bind` \a ->
               unit (Just a))
    =  lift (m `bind` \a -> k a)
```

Third monad transformer law

```
unlift (lift m)
    =  unMaybeM (lift m) `bind` \maybea ->
       case maybea of
```

```
        Just a  -> unit a
        Nothing -> error "..."
    =  unMaybeM (MaybeM (m `bind` \a ->
                          unit (Just a))) `bind` \maybea ->
       case maybea of
         Just a  -> unit a
         Nothing -> error "..."
    =  m `bind` \a ->
       unit (Just a) `bind` \maybea ->
       case maybea of
         Just a  -> unit a
         Nothing -> error "..."
    =  m `bind` \a ->
       case (Just a) of
         Just a  -> unit a
         Nothing -> error "..."
    =  m `bind` \a ->
       unit a
    =  m
```

The `StateT` monad transformer

First monad transformer law

```
lift (unit a)
    =  StateM (\s -> unit a `bind` \x -> unit (s,x))
    =  StateM (\s -> unit (s,a))
    =  unit a
```

Second monad transformer law

```
lift (m `bind` \q{a} k a)
    =  StateM (\q{s} (m    `bind` \q{a}
                      k a) `bind` \q{x}
                     unit (s,x))
    =  StateM (\q{s} m          `bind` (\q{a}
                     k a        `bind`  \q{x}
                     unit (s,x)))
    =  StateM (\q{s} m          `bind` \q{x}
                     unit (s,x) `bind` \q{(s1,a)}
                     k a        `bind` \q{x}
                     unit (s1,x))
    =  StateM (\q{s} m          `bind` \q{x}
                     unit (s,x) `bind` \q{(s1,a)}
                     unStateM (StateM (\q{s} k a `bind` \q{x}
                                             unit (s,x))) s1)
    =  StateM (\q{s} m          `bind` \q{x}
                     unit (s,x) `bind` \q{(s1,a)}
                     unStateM (lift (k a)) s1)
    =  StateM (\q{s} m `bind` \q{x}
                     unit (s,x)) `bind` (\q{a}
```

```
                    lift (k a))
    =  (lift m) `bind` (\q{a} lift (k a))
```

Third monad transformer law

```
unlift (lift m)
    =  (unStateM (lift m)) undefined `bind` \(s,a) ->
       unit a
    =  (unStateM (StateM (\s -> m `bind` \a ->
                                unit (s,a)))) undefined `bind` \(s,a) ->
       unit a
    =  (\s -> m `bind` \a ->
              unit (s,a)) undefined `bind` \(s,a) ->
       unit a
    =  m `bind` \a ->
       unit (undefined,a) `bind` \(s,a) ->
       unit a
    =  m `bind` \a ->
       unit a
    =  m
```

Appendix B

Evolving algebra specifications

The syntax of evolving algebra specifications accepted by EVADE, is given by the following syntax description:

specification → MODULE name parameterlist signature
start program stopreturnexpr

parameterlist → [(parameter (, parameter)*)]

returnexpr → term : typeexpr

signature → staticsorts dynamicsorts staticfunctions dynamicfunctions

start → START updateset

program → (transitionrule)*

stop → STOP condition

parameter → name : typeexpr

staticsorts → SS (name ==> gofertype)*

dynamicsorts → DS (name)*

staticfunctions → SF (parameter ==> gofertype)*

dynamcfunctions → DF (parameter)*

transitionrule → TRANSITION name IF condition THEN updateset

updateset → (update)*

update → simpleupdate | newupdate

simpleupdate → updatableterm := term

newupdate → NEW updatableterm : name WITH updateset .

updatableterm → name [(term (, term)*)]

condition → term | (condition) NOT condition |
term termrelation term | DEFINED (term) | TRUE | FALSE |
condition conditionoperator condition

term → name [(term (, term)*)] | (term) | - term |
number | term termoperator term

termoperator → + | - | * | /

termrelation → = | /= | < | > | <= | >=

conditionoperator → /\ | \/

typeexpr → name | (name (, name)*) -> name

Bibliography

[Bar84] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. North-Holland, revised edition, 1984.

[BP95] Bernhard Beckert and Joachim Posegga. leanEA: A poor man's evolving algebra compiler. Interner Bericht 25/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[BR94] Egon Börger and Dean Rosenzweig. A mathematical definition of full prolog. In *Science of Computer Programming.* (to appear), 1994.

[Die95a] Stephan Diehl. Transformations of evolving algebras. Technischer Bericht A 02/95, Universität des Saarlandes, Computer Science Department, 1995.

[Die95b] Dag Diesen. Specifying algorithms using evolving algebras. implementation of functional programming languages. Research report 199, Department of Informatics, University of Oslo, March 1995.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[Gur91] Yuri Gurevich. Evolving algebras, a tutorial introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264 – 284, February 1991.

[Gur95] Yuri Gurevich. Evolving algebras 1993; lipari guide. In Egon Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[HJe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HM] Jim Huggins and Raghu Mani. The evolving algebra interpreter version 2.0.

[Hug95] James K. Huggins. An offline partial evaluator for evolving algebras. Technical Report CSE-TR-229-95, University of Michigan, EECS Department, 1995.

[Jon] Mark P. Jones. An introduction to gofer. Available from http://www.cs.yale.edu/.

[Kap93] Angelica Maria Kappel. Executable specifications based on dynamic algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 689 of LNAI. Springer, 1993.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995.

[PH94] Arnd Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume Volume I: Technology/Foundations, pages 434 – 439. Elsevier, 1994.

[Ton93] Hans Tonino. A formalization of many-sorted evolving algebras. Report 93-115, Delft University of Technology, Faculty of Technical Mathematics and Informatics, 1993.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms*. International Series in Computer Science. Prentice Hall, 1990.