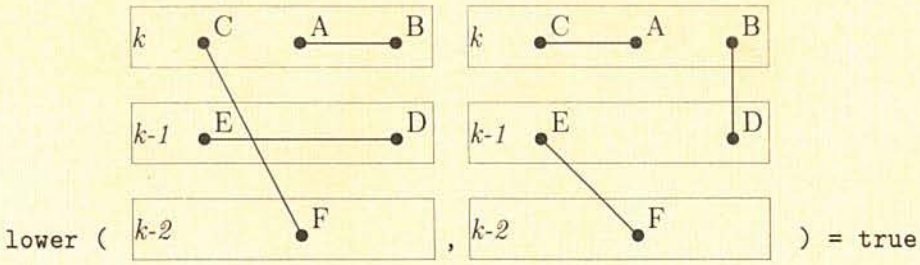
MODULAR ALGEBRAIC SPECIFICATIONS AND TRANSFORMATIONAL PROGRAM DEVELOPMENT

N.W.P. van Diepen



Stellingen
behorend bij het proefschrift

Modular Algebraic Specifications and
Transformational Program Development

Niek van Diepen
12 oktober 1994

1. Met observationele semantiek kan een voor hergebruik geschikte granulariteit van programmatuur bereikt worden. [Hoofdstuk 4]

2. De gangbare weekendtoernooien op basis van het Zwitsers systeem bij schaken zouden aan spanning winnen door de toepassing van het MacMahon-systeem. Tevens zou de veel toegepaste kunstmatige scheiding in twee groepen kunnen vervallen, en kan men met minder ronden toe. [Hoofdstuk 5]

3. Object-oriëntatie staat in dezelfde verhouding tot modularisatie als gestructureerd programmeren tot programmeren met sprongopdrachten: door de opgelegde regels is het veel eenvoudiger met een goede oplossing te komen, maar sommige optimale oplossingen zijn niet meer mogelijk.

4. Pogingen tot specificatie van de werkelijkheid resulteren bij programmeren in steeds dikkere vereistendocumenten, net als bij conflictsimulaties in steeds dikkere spelregelboeken, zonder dat het doel echt dichterbij wordt gebracht. 'Virtual reality' zal aan het zelfde euvel blijven lijden, hoe ver de computertechnologie ook voortschrijdt.

5. De `LEAVE`-constructie in de programmeertaal ELAN kan beter uit het eerste semester vertrekken.

6. Het gebruik van een schreefloos lettertype maakt het vrijwel onmogelijk onderscheid maken tussen de hoofdletter I en de kleine letter l. Weergave in drie verschillende diktes helpt daarbij niet. [Gordon 1979]

7. Door het voeren van de campagne *Kies Exact*, gecombineerd met het ongeveer gelijktijdig invoeren op het VWO van een wiskundevak dat geen B-studie toelaat, is het percentage meisjes dat een B-studie aanvangt van de meisjes die aan wetenschappelijk onderwijs beginnen niet significant toegenomen. [Zakboek onderwijsstatistieken, div. jg.]

8. De inrichting van de Aula Maior aan de Katholieke Universiteit maakt het moeilijk alle bij de promotieplechtigheid betrokkenen van een geschikte plaats te voorzien.

9. De FIDE (Wereldschaakbond) had bij de uitbreiding van de 50-zetten-regel van de ervaringen van de Nihon Ki-in (Japanse Go-Bond) op het gebied van uitzonderingsregels kunnen leren. Gelukkig zijn beiden inmiddels op hun schreden teruggekeerd.

10. Bij de meeste analyses van operatie Market-Garden wordt vergeten dat het Roergebied het uiteindelijke doel was, en Arnhem slechts halverwege. Er zou na Arnhem nog minstens één brug verder moeten worden getrokken.

Modular Algebraic Specifications and Transformational Program Development

een wetenschappelijke proeve op het gebied
van de
Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
woensdag 12 oktober 1994,
des namiddags te 3:30 uur precies

door

Nicolaas Wilhelmus Petrus van Diepen

geboren op 12 maart 1959 te Wormer

Druk: Febodruk – Enschede

Promotores: Prof.dr. H.A. Partsch

Prof.dr. P. Klint
(Universiteit van Amsterdam)

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Diepen, Nicolaas Wilhelmus Petrus van

Modular algebraic specifications and transformational
program development / Nicolaas Wilhelmus Petrus van
Diepen. - [S.l. : s.n.] (Enschede : Febodruk). - Ill.
Proefschrift Nijmegen. - Met lit. opg.
ISBN 90-9007438-4
Trefw.: transformationeel programmeren / modularisatie /
algebraïsche specificaties.

Contents

Contents iii

Preface vii

1 Introduction 9
- 1.1 Why Formal Software Development? . . . 9
- 1.2 Formal Specifications . . . 11
- 1.3 What are Modules? . . . 12
- 1.4 Algebraic Specifications . . . 14
- 1.5 Modules and structuring . . . 15
- 1.6 Modules and Transformations . . . 18
- 1.7 Modules and Reuse . . . 18
- 1.8 Efficient implementation and semantics . . . 22
- 1.9 Disadvantages of formal specifications . . . 23
- 1.10 An overview of the thesis . . . 24

2 Formalizing Informal Requirements – Some Aspects 27
- 2.1 Introduction . . . 27
- 2.2 Requirements specification . . . 28
  - 2.2.1 A practical example . . . 29
  - 2.2.2 Different Kinds of Requirements . . . 33
  - 2.2.3 Desirable Properties of Formalisms for Requirements Definitions 35
  - 2.2.4 How to Proceed . . . 37
- 2.3 Formal Specification . . . 38
- 2.4 The Process of Formalization . . . 40
  - 2.4.1 Problem Identification . . . 40
  - 2.4.2 Problem Description . . . 45
  - 2.4.3 Analysis of the Problem Description . . . 48
  - 2.4.4 Structuring . . . 50
- 2.5 Conclusions . . . 51

3 SMALL – dynamic semantics of a language with GOTOs 53
- 3.1 The specification of jump-statements . . . 53
- 3.2 Syntax and informal semantics of SMALL . . . 55
  - 3.2.1 Abstract syntax and informal semantics . . . 55
  - 3.2.2 Concrete syntax . . . 57

3.3 Algebraic semantics of the SMALL kernel 59
3.3.1 The environment 59
3.3.2 The dynamic semantics of SMALL1 61
3.4 SMALL with GOTOs 66
3.5 A note on modularity 70
3.5.1 Auxiliary functions 70
3.5.2 An alternative definition of SMALL1 70
3.5.3 A problem with extendint the alternative SMALL1 definition . 72
3.5.4 The alternative definition of SMALL2 72
3.6 Implementation of the SMALL specification 75
3.6.1 An "ad hoc" implementation 75
3.6.2 The automatic scheme of Drosten and Ehrich 77
3.6.3 Outermost reduction strategies 78
3.7 Conclusions 80

4 Implementation of Modular Algebraic Specifications 81
4.1 Introduction 82
4.1.1 Modular algebraic specifications 82
4.1.2 Implementation of algebraic specifications 83
4.1.3 Related work 83
4.1.4 An outline of this Chapter 85
4.2 The formalisms informally 86
4.2.1 Horizontal composition in algebraic specifications 86
4.2.2 Hoare logic and abstract data types 86
4.3 Algebraic implementation 88
4.3.1 Initial algebra semantics and reusability 88
4.3.2 A theory of algebraic implementation 92
4.4 An example: Tables 102
4.4.1 Definition (well-formedness of search trees) 105
4.4.2 Well-formedness lemma for trees 105
4.4.3 Proof of `lookup`-equality 106
4.5 Functional implementation 108
4.5.1 The functional view 108
4.5.2 A theory of functional implementation 109
4.6 An example: Tables revisited 114
4.7 Conclusions 118

5 From Formal Specification towards Derivation: the MacMahon (Swiss) System 119
5.1 Introduction 120
5.1.1 The Swiss System informally 120
5.1.2 A difficult problem 121
5.1.3 An overview of the Chapter 121
5.2 The informal description 122
5.3 The formal specification 123

5.3.1 The basic datatypes for a tournament 123
5.3.2 The pairing specification 127
5.4 Derivations at the specification level 130
5.4.1 The need for specialization 130
5.4.2 Domain knowledge 131
5.5 Some possible variations in the pairing system 134
5.5.1 Random draw 134
5.5.2 Special circumstances 134
5.5.3 Go tournaments: the MacMahon system 135
5.6 Implementing the specification 137
5.6.1 The case study: MacMahon tournaments 137
5.6.2 Correctness and weight validation 138
5.6.3 Speed 138
5.7 Conclusions 140
Annexe: Selected parts of the pairing algorithm 142

Bibliography **147**

Summary **155**

Samenvatting **157**

Curriculum Vitae **161**

Preface

Writing a Thesis is a long-winded job. A somewhat faster colleague recently called his Thesis "It's About Time" [Thi94]. In the same vein I could have called this Thesis "The slow road to optimization". Computers have improved in speed and size by an order of magnitude while I worked on it. So just waiting a few more years would be enough for computer technology to overtake some of the improvements made here.

Fortunately the picture is not so bleak. Faster computers make more complicated applications possible, and more complicated applications need better program structuring and specification, the other main subject of the Thesis. And I still feel that improving crucial parts of programs will remain an important theme. This Thesis provides a technical solution for the separate treatment of those crucial parts.

Work on this Thesis has started at the Centre for Mathematics and Computer Science (CWI) in Amsterdam, as part of the ESPRIT-I Project 348 (Generation of Interactive Programming Environments – GIPE). I was at that time also briefly involved in ESPRIT-I Project 432 (An Integrated Formal Approach to Industrial Software Development – METEOR). Chapter 3 (formerly appearing in [BHK89]) and the original work on Chapter 4 have been done at the CWI. Support from the European Communities and the Netherlands Organization for Scientific Research (NWO – CWI is an NWO Institute) are gratefully acknowledged.

The Thesis has been finished at the Katholieke Universiteit Nijmegen as part of the NFI project STOP – Specification and Transformation Of Programs – under grant NF 63/62-518. Chapters 1, 2 (written together with Prof. Partsch, and published before as [DP91]), and 5 have been written here, and Chapter 4 has been improved (an extended abstract appeared as [Die88] – the current improved version is submitted for publication). Support from the Netherlands Organization for Scientific Research (NWO) is gratefully acknowledged again.

Many people over the years have contributed to this Thesis in one way or another. When it comes to technical support some stand out more clearly than others. No attempt at completeness is made. I would like to thank my roommates at the CWI and the KUN, especially Jan Heering, Ard Verhoog, Paul Hendriks, Norbert Völker and Max Geerling, and the secretaries Greta Löw and Ineke Kuster, for their positive atmosphere. Let them be a symbol for my many colleagues over the years. I would also like to thank my bosses at various stages: Paul Klint and Jan Heering in Amsterdam, and Helmut Partsch and later on Hans Meijer in Nijmegen. Thanks are also due to the Manuscript Commission (Professors Bernhard Möller, Jos Baeten and Kees Koster), and to the anonymous referees of *Acta Informatica*, who will find

their suggestions on Chapter 4 in print here for the first time. Finally, my students kept me on my toes, and special thanks go to the GoMMTour group: Araminte Bleeker, Michiel Boudewijns, Martijn Evers, Caspar Scholten, Joost Schoonderbeek, Patrick Toonen, Theo Wiggerts, and their supervisors Paul Frederiks, Wil Lamain, and Harrie van Seters for providing the environment to apply the algorithm from Chapter 5.

When a project takes so long moral support is desperately needed, especially during the long period when it is 'almost finished'. I would like to thank my friends from the Nijmegen games club 'Casus Belli' and the Nijmeegse and Utrechtse Go Clubs for their nagging inquiries at times, and for a way to relax. Even more nagging (in a friendly but firm way) was Gwen van der Velden, who pushed me along for more than a year. And the last year, traditionally the hardest by far, got a light touch and warm glow due to Annemarie Hovingh.

Undoubtedly I would not have seen the end of this tunnel without the support and positive attitude towards learning provided by my parents. I do not want to underrate the impulse provided by my father in any way, but I dedicate this Thesis to my mother. She could have done it herself, given half my opportunities. Unfortunately she was the oldest daughter in a large family at the wrong time.

Niek van Diepen
Nijmegen, July 1994

To Thea van Diepen-Tap

Chapter 1

Introduction

1.1 Why Formal Software Development?

This thesis deals with one technique to handle the complex task of programming: first obtain a formal specification and then derive a program from this specification via semantics preserving transformations. Hence, if the specification is correct, then by construction the program is correct.

Complexity

Key to any program construction methodology is controlling problem complexity, especially with respect to problem size and to the correctness of possible programming solutions. Most programs have to run on complex data, producing non-trivial results. Still we have to be sure that each result in itself is correct. W.r.t large programs neither verification (correctness proofs) nor validation (testing) have been successful in eliminating all mistakes.

Small programs, on the other hand, can be understood and shown to be valid, and probably even proven correct directly. So if we can reduce our problems to small enough sub-problems these sub-problems can be solved. Also, program derivation (allowing for correctness proofs by construction) becomes feasible. Then the results should be combined in such a way that the combination process is of a lower complexity than the original problem, much like the derivation of a Theorem from several Lemmata.

Related Approaches

Program transformation is by no means the only method to write correct programs in an efficient way. Some related approaches are:

programming techniques

- The first systematic technique, *structured programming*, met with limited success. Each technique has its limitations, and of course some problems are

hard to program either top-down or bottom-up. The main problem though, is that problem decomposition is not well supported. The top-down approach attempts to provide a decomposition, and if it fails no remedy is available. Hence programming this way is essentially still a one person job. The programs of today are too large for that, needing teams of programmers working together to be finished in a reasonable amount of time.

- In recent years *object-oriented programming* (OOP) seems to sweep the field. The essence of this technique is the central role of objects: all functions are grouped around the objects they use, thus automatically supporting information hiding. Also, OOP supports inheritance of properties from simple objects to more complex objects derived from them, thus reducing the implementation effort to a correct rendering of the original object.

  However, the starting point for OOP is an object-oriented specification. Modular algebraic specifications as described in this thesis are object-oriented if each module contains exactly one sort and all of its associated functions. Hence OOP can be seen as a special case of the approach presented here.

specification-based techniques

- *Reduction of the problem area* to some well-known problem domain (e.g., information systems, compiler construction) allows for development of dedicated specification and programming tools. Within such a domain additional knowledge is available, so specification techniques can be developed which take advantage of this knowledge.

- Production of *executable specifications*, e.g., via rewrite systems, reduces the complexity of the implementation phase to a trivial process. However, producing such a specification usually is just a different form of programming, and hence the complexity of the specification process is higher than strictly necessary.

To summarize: both structured and object-oriented programming provide ways to achieve a complexity mastering division of labour. Reduction of the problem area allows the development of more powerful basic tools geared towards the particular problem area. These tools in themselves can be quite complex, but as unitary building blocks for larger programs they encapsulate this complexity. Only a 'user guide' is needed. Executable specifications allow the programmer to focus on the specification, and forget about the internal complexity of these building blocks.

Why modular algebraic specifications?

Modular algebraic specification allows for more general types of structuring than the object-oriented and the top-down approaches. It can also allow for automatic execution if the specification is of the right form. So these approaches may be seen as special cases of the approach chosen here, while at the same time the problem

area is not restricted. But even dedicated software may well be specified in modular algebraic specifications, since the technique presented in Chapter 4 makes optimal dedicated implementations possible. So the best of all these techniques can be combined by the use of algebraic specifications. The restriction on the use of functions implemented this way is a small price to pay.

The next sections describe how modular algebraic specifications can contribute to structuring of the requirements in such a way that each module is of reasonable complexity. If this structure can be retained upto the implementation the final program will remain of reasonable complexity.

1.2 Formal Specifications

The central theme of this thesis is the use of modular algebraic specifications in programming. Motivation for the use of such specifications in programming will be given in the remainder of this Chapter. This section aims at motivating the choice for formal specifications.

An overview of the different specification methods currently in use is beyond the scope of this thesis. The interested reader can find an annotated bibliography in [COMPASS91]. For instance, a strong case can be made for graphical support of the specification process, and this is a non-trivial task for a text-oriented system like the specification formalism used here. On the other hand, graphically oriented systems are more difficult to validate via a computer at the moment.

Why formal specifications?

Historically, natural language is our first choice of vehicle for any specification. It is surely versatile enough, anything that we want to be specified can be expressed in natural language. If needed the language can even be extended on the fly. Language can be precise enough, even if this requires hard and careful work on definitions. And no special training is needed to understand natural language.

So why should we bother with formal specifications at all?

- Enhanced *quality control* is made possible via formal language. Consistency (nothing is specified in a contradictory way) and completeness (nothing is left out of the specification) are the two most important properties of a specification meant for later implementation. The use of a formal language helps in ascertaining these properties, since the occurence of different references to the same item is restricted. Even automatic tools for validation (like type checkers) may be made available.

- Formal specifications are the key to *program transformation*. To be able to show the preservation of correctness of the implementation a starting point is needed. This must be a formally well-defined description of the valid set of implementation models. Such a description is only practical in a formal specification language.

```
module Booleans
begin
  exports
    begin
      sorts BOOL
      functions
        true :              -> BOOL
        false:              -> BOOL
        and  : BOOL # BOOL -> BOOL
        or   : BOOL # BOOL -> BOOL
        not  : BOOL        -> BOOL
    end

variables
  b: -> BOOL

equations
[b1] and(true, b)  = b
[b2] and(false, b) = false
[b3] or(true, b)   = true
[b4] or(false, b)  = b
[b5] not(true)     = false
[b6] not(false)    = true

end Booleans
```

Figure 1.1: a basic module

- A formal specification is needed for *proofs* of any kind. If a specification is only available informally, plausibility arguments are possible, but a rigorous mathematical proof will lack foundation.

1.3 What are Modules?

This section introduces the key structuring technique used in this thesis: the *module*. For an intuitive introduction to this concept have a look at the basic module `Booleans` in Figure 1.1. It is written in the Algebraic Specification Formalism ASF [BHK89], which will be used throughout this thesis.

This module specifies only one sort (more sorts or even none are also possible), `BOOL`, with the familiar functions `and`, `or` and `not`. The constants `true` and `false` are seen as functions without arguments (they give the same result each time). For convenience they are written as `true` instead of the strictly speaking more correct `true()`, a function with an empty parameter list. These functions and constants

are in the *export* section, which means that when this module is used they will all be available. How a module can use (called *import*) other modules will be explained in section 1.5. In section 1.7 the *hidden* section will be introduced to be able to distinghuish sorts and functions not meant for use outside their defining module.

Semantics

To be able to discuss the meaning of the specification in Figure 1.1 we need to know what we have specified. A model satisfies a specification when it contains representatives for all sorts, functions and constants in the specification, and when these representatives satisfy the equations.

The usual semantics are defined by the *term model*: the set consisting of all correctly typed terms which can be formed with these functions. E.g., `and(true, and(false, true))`, a term with two arguments, one of which is another correctly typed term with two arguments. Incorrectly typed terms, e.g., `or(true, false, true)`, do not exist in this model.

The *equations* now define which terms are equal to (in other words, may be replaced by) which other terms, e.g.:

```
not(and(or(true,false),and(false,true))) =          (equation b2)
not(and(or(true,false),    false      )) =          (equation b3)
not(and(   true       ,    false      )) =          (equation b1)
not(                       false       ) =          (equation b6)
                           true          =          (equation b4)
              or(    false,true)          =          (equation b5)
              or(not(true),true)
```

Note that the equations can be used in both directions, not just from left to right, as the use of equations b4 and b5 above illustrates. Terms which are not equal according to these equations are not equal in the term model. The term model together with the equations now define the *initial semantics* of this algebraic specification.

The term model is a representative of an initial model. *Initial* means that for other models a homomorphism will always be available from the initial model to this other model. Models are often non-initial, though. An important example of a usually non-initial model is the set of Booleans of the programming language used to implement this specification. It might also contain an *undefined* value, which has no equivalent in the initial model, or extra functions like the `xor`.

Intuitively, the least complex term (`true`) can be seen as the 'answer', the so-called *normal form*, of the terms in the example. In principle though, this term is only one of many equal terms. Indeed the set of normal forms is not necessarily fixed by the specification. Profitable changes in the implementation may result from the judicious choice of sets of normal forms, as will be shown in Chapter 4.

Modules

To summarize, a module consists of a definition of *sorts* and *functions* (divided into *export* and *hidden* parts – for the latter see section 1.7), and an *equations* part (with variables) defining the functions. Optional parts are the *import* section and a *parameters* part – these are treated below.

1.4 Algebraic Specifications

The module `Booleans` is an example of an algebraic specification. This section briefly reviews the theory.

Algebraic structures are among the most versatile of mathematical structures. The underlying principles are well-understood. Through the ages, algebraic manipulation has been smoothed to a stable and easy to use technique which is useful in a wide variety of application domains. Hence specifications in the form of an algebraic structure are recognizable to everybody with a basic mathematical background.

The examples in this Chapter provided some insight in the algebraic specification format used in the thesis. For a detailed description of the semantics of ASF the interested reader is referred to Chapter 1 of [BHK89]. The essential parts of an algebraic specification correspond to the structure of the specification in the preceding section:

- A set of *sorts*, usually called S with elements often denoted as s_i or s. Of course, S can (and in mathematics often does) contain one sort only. Specifications of programs almost always include more than one sort. A specification with an ordering, for instance, will also contain the Booleans.

- A set F of *functions* and *constants* over those sorts. Constants are seen here as functions without an argument. Alternatively they could form a separate set.

- Some description method for the *semantics* of the functions: a set of rules, in this thesis often a set E of *conditional equations*. However, other logical systems, e.g., Horn Clause Logic or rewrite systems, are possible. In Chapter 3 it will be discussed how rewrite systems may be generated from the equations.

- A set of *valid models*. Often the choice is restricted to the initial or to the final model, because they are unique up to isomorphism if they exist. But for optimization purposes it will be desirable to allow just any model validating the equations in the specification.

Remarks

- To allow for implementability only finite sets of sorts will be taken into account. Generic sorts will still be allowed when they can be seen as a finite number of instantiations. Then each version of the generic sort may be seen as a distinctive sort. Hence the number of sorts is still finite.

- Again, a finite set of functions is desirable for implementation reasons. Due to the finiteness of the set of sorts generic functions (i.e. functions with arguments with indeterminate type) are possible. The choice for each argument type is finite, hence the number of different functions with the same function name generated in this way is finite too.

- For ease of understanding usually the initial model is chosen, not only because of the uniqueness up to isomorphism, but also because there is a constructive way to generate an initial model, so we can be sure it exists. This constructively available model is the *term model* (see section 1.3).

1.5 Modules and structuring

Assume that we have a method to structure programs in some way. What do we want to do with it? And what kind of demands do these wishes place on the structure involved?

granularity The structure should be flexible enough to make small components, organized around one sort or one function, i.e. small granularity. Then a profitable focus of attention for optimization can be reached. The *modules* in ASF presented here allow for this.

inheritance With small modules only relatively trivial types can be described, hence one should be able to build bigger modules. While building, the properties of the smaller modules must be amalgamated in the bigger module. Those constructed modules in turn can then be used to build even bigger modules, and so on. The combination of properties via layers of modules is called *inheritance*. So an effective inheritance mechanism, working on the import-export-part (see below) of the specification, is also needed. ASF [BHK89] uses a rather simple normalization scheme, which combines all module specifications with proper renaming to form one huge module, containing all functions and sorts in the specification.

parameterization Instead of direct inheritance, often the same structure must be specified for different data types. It is convenient to be able to make one specification only, *parameterized* with a sort and its associated functions. This is often a sort with certain properties, e.g., an ordering. This is provided for in ASF, and an example can be found in Chapter 3.

import-export The focus of attention per module advocated before is lost if activities within one module also change objects outside this module, e.g. when updating an external database. A similar problem exists when activities from the outside have effect on the inside. In both cases one has to view the whole system before the semantics, and hence correct implementations, can be determined for individual modules, since changes in one place may force changes elsewhere.

So a border mechanism is needed: not everything happening within the implementation of the whole system should be allowed to affect the semantics of the module — only specifically designated, so-called 'imported', sorts and functions. Similarly not everything should be 'exported', only those things really needed on the outside. Hence we want an effective *import-export* mechanism for modules. The normalization process within ASF combines everything, hence in the combined module those borders do not automatically exist. Explicit renaming has to ensure separation.

hiding A description in formal semantics runs the risk of overspecification, seemingly showing design decisions where there are none. E.g., a storage structure may be specified as a linked list because such a structure is quickly written down. However, while this structure may be convenient for a quick specification, it can be irrelevant or misleading. Then it can become harmful because it does not allow for other, possibly more efficient, implementations.

Within ASF a *hiding* mechanism is provided to allow the specification writer to distinguish between necessary parts of the specification and convenient detail. Allowing declarative (i.e. non-operational) specifications only is the main alternative to eliminate this unnecessary detail. However, this restricts the specifier, and hence it is often unpleasant to work with. Other descriptions of the semantics, e.g., via predicates, may be more difficult to reason with. It will be argued in the next section that some sort of formal semantics is needed, though.

Another example: the Naturals

An example of an importing specification is the module `Naturals` in Figure 1.2. The constants `true` and `false` (and also the functions `and`, etc., from module `Booleans`, which are not used) do exist in the module `Naturals`, and (automatic in ASF) in any module importing `Naturals` in turn. The *normal* forms (intuitively the 'answers' or 'results') now include the terms of the form `0`, and `succ(0)`, `succ(succ(0))`, etc., and all normal forms from `Booleans` (because for terms containing `eq` this function can always be eliminated using the equations, no extra elements of sort `BOOL` are formed, so the old normal forms are sufficient).

Semantics

An interesting twist occurs when this specification is imported in some module and then extended with a minus function `minus:NAT # NAT -> NAT`. The term `minus(0,succ(0))` (usually written as -1) is not in the initial model of the Naturals since it has no equivalent in the term model. The following choices now exist:

- disallow this function, since it generates new terms;
- put in an error value; or
- accept that those new terms have to be modeled too.

```
module Naturals
begin
  exports
    begin
      sorts NAT
      functions
        0   :            -> NAT
        succ: NAT        -> NAT
        plus: NAT # NAT -> NAT
        mul : NAT # NAT -> NAT
        eq  : NAT # NAT -> BOOL
      end

  imports Booleans

variables:
  m, n: -> NAT

equations:
[n1] plus(0,n)           = n
[n2] plus(succ(m),n)     = succ(plus(m,n))
[n3] mul(0,n)            = 0
[n4] mul(succ(m),n)      = plus(n,mul(m,n))
[n5] eq(0,0)             = true
[n6] eq(0,succ(n))       = false
[n7] eq(succ(m),0)       = false
[n8] eq(succ(m),succ(n)) = eq(m,n)

end Naturals
```

Figure 1.2: a specification of the natural numbers

All three approaches have their uses and they are possible within the framework of algebraic specification.

Notation

When no confusion arises from the context the more readable notations `a AND b`, `a OR b`, `NOT a`, `a + b`, and `a * b`, may be used rather than `and(a,b)`, etc. Similarly `succ(0)` will often be written as 1, `succ(succ(0))` as 2, etc.

This syntactic sugar is no real extension of the functions and constants allowed in ASF. It is even made formal within the SDF syntax definition formalism on top of ASF ([BHK89], [HHKR89]).

1.6 Modules and Transformations

The power of this form of modular specifications, especially the possibility to hide parts of the specification, now allows us the use of a strong technique to improve the execution behaviour of individual functions (or groups of functions on one abstract data type): *transformational derivation*. The power of this technique is demonstrated, using Hoare Logic as a vehicle, in [DR86]. The link between Hoare Logic proofs and Algebraic Specifications is given in Chapter 4, where it is shown how these approaches can be effectively combined.

Algebraic specifications themselves can be subjected to transformational derivations (see [Par90] for examples and [COMPASS91] for an extensive bibliography). Also, it will be argued in Chapter 5 that derivations at the specification level can be quite useful.

1.7 Modules and Reuse

Another problem to which modularity provides a solution is *software reuse*, an as yet underdeveloped field. Once a good piece of program has been written it is a waste of time to program a similar piece again as part of another system. That would quite possibly introduce old errors and real improvement is hardly to be expected. This programming using larger building blocks is an old and promising idea, but it has hardly delivered yet, except for restricted areas of application.

Inhibitions to reuse

Why is software reuse not more popular? Apart from purely human factors (programmers are convinced they will produce a better piece of code than their peers) the main problem is one of software classification: how can an appropriate piece of software be found?

Obviously, a specification is needed. Once it is described in full what a software module is supposed to do, it can be indexed and hence retrieved later. It will be shown in this thesis that modules with the same interface but different run-time behaviour can be developed. If this behavioural difference can be described

(probably by the original implementer) the re-user can make a choice between the different implementations available to him or her, based on the properties needed in the new program.

Different implementations

To give an example of a specification which has clearly different but equivalent implementations (i.e., they satisfy the same specification) we turn to the module `Sets-of-Naturals` in Figure 1.3. (In practice one would make a parameterized module `Sets-of(M)`, but the added technical complications would clutter the issue under discussion here.)

Many 'standard' implementations of sets exist, e.g., lists, search-trees, arrays, hash-tables, or bit-rows. Each of those has its own advantages and disadvantages. For instance, arrays, tables and bit-rows have in common that they only allow for limited finite and relatively small-sized sets. If the maximum size is large enough the user may never find out what the actual limit is, so the external behaviour (the results of applying the functions) will be the same for all implementations with average use. The same cannot be said about efficiency, so a real choice is available.

Evaluating different implementations

When trying to evaluate efficiency it is convenient to look at the exported functions, since these have to be implemented: `emptyset`, `add`, `delete`, `is-in` and `size`. The hidden function `list` will not be visible in the actual implementation — it will have been replaced internally in all options. The table in Figure 1.4 gives the expected average efficiency.

No attempt has been made in the table to account for optimizations specific to the problem domain. Some of those specialized improvements are possible, e.g., if `size` is important a size counter may be added to the implementations above (the array implementation already has such a counter). Also, the hash-table figures depend on some expected distribution of the elements in the set to be really effective, but we assume some suitable distribution for the moment. It is only necessary for the pursuit of the argument in this section.

For most structures the entries are obvious, but search-trees exist in many shapes, and not all support deletion of information. The table-entry $\mathcal{O}(\log n)$ can be achieved by looking up the information and marking it as deleted. It is often the best result in practice. However, in the implementation taken as an example here deleted values are only 'marked as deleted'. Hence they still exist and may end up cluttering the structure with many redundant nodes. So this technique may be used to the detriment of the time needed for addition and member-query. A smarter deletion function can be implemented, which will raise the deletion costs considerably. In the worst case rebuilding of the search-tree may be needed (n entries, so the deletion cost may grow to $\mathcal{O}(n \log n)$).

```
module Sets-of-Naturals
begin
  exports
    begin
      sorts SET
      functions
        emptyset:            -> SET
        add      : NAT # SET -> SET
        delete   : NAT # SET -> SET
        is-in    : NAT # SET -> BOOL
        size     : SET       -> NAT
    end

  hidden functions
    list: NAT # SET -> SET

  imports Booleans, Naturals

variables
  s  : -> SET
  n,m: -> NAT

equations
[s1]  is-in(n,emptyset)   = false
[s2]  is-in(n,list(n,s))  = true
[s3]  is-in(n,list(m,s))  = is-in(n,s)
        where eq(n,m) = false
[s4]  add(n,s)            = s
        where is-in(n,s) = true
[s5]  add(n,s)            = list(n,s)
        where is-in(n,s)  = false
[s6]  delete(n,emptyset)  = emptyset
[s7]  delete(n,list(n,s)) = s
[s8]  delete(n,list(m,s)) = list(m,delete(n,s))
        where eq(n,m) = false
[s9]  size(emptyset)      = 0
[s10] size(list(n,s))     = size(s) + 1

end Sets-of-Naturals
```

Figure 1.3: sets of natural numbers

implem-entation	use of memory	functions				
		`emptyset`	`add`	`delete`	`is-in`	`size`
list	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
search-tree	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
array	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
hash-table	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$
bit-row	$\mathcal{O}(m)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$

n is the current number of elements, m the maximum (where applicable)

Figure 1.4: average efficiency of set implementations

How to choose

Now this specification, describing some members from a (much larger) family of implementations gives a stable interface for potential reusers. The actual choice for one implementation above the others depends on more factors, like memory usage and compatibility with other programs.

However, Figure 1.4 addresses an important criterion for the evaluation needed for the choice of the best implementation in any particular case of reuse. Suppose not many deletions or member queries are expected, but insertion must be fast, then lists may be the best implementation. If insertions, deletions, and member queries are frequent, bitrows may be ideal — that is, if memory (constant size of the implementation) or correctness (the minimum and maximum value of the integers in the set will be limited by the implementation) considerations do not prohibit their use.

Prerequisites for reuse

Another necessary, but not sufficient, condition for reuse is self-containedness of the piece of software intendend for reuse. This is seldom fully achieved (Boolean and arithmetic operators usually come from the library of the programming language used for implementation). However, if library use is the only use of externally implemented operations implementation details of these external operations will almost always be the same, so dependence on these details will be of acceptable level per module.

Not so easily realized is the adaptation of a module to new, possibly unexpected, demands. This tailoring to the new demands may be next to impossible if the source code is not available (for Reverse Engineering, see, e.g., [KK91]). A re-implementation of the adapted specification may already provide some improvement in the programming process. But ideally the derivation of the various implementations is available. Then re-implementation via program transformations (see below) may deviate from the old development at a convenient place.

In summary, modular specifications and implementations make software reuse feasible by providing a self-contained, at least partly documented implementation, or ideally a choice of implementations. Extra work from the implementor is needed to provide run-time behavioral information, and some educational effort in software

reuse is probably needed, but this is outside the scope of this thesis.

Reuse and transformations

Transformational derivation offers the tools needed to provide a wide choice of different implementations. If also a description of the derivation process is available other implementations may be generated with relatively little effort.

At the same time adherence to the specification will guarantee a stable interface and sufficient self-containedness. Different derivations from the same specification are provided in [DR86], while a formal treatment of one interface specification for different implementations can be found in Chapter 4.

1.8 Efficient implementation and semantics

For efficient implementations often the initial semantics are too restrictive. One wants to have more freedom by allowing other than the specified functions (technically more than asked for, these are also known as '*junk*'), since a better implementation can be given using them. If they occurred within the specification they would be hidden, of course, but even non-specified functions are possible in the final implementation. Also, if the results of some functions are not influenced by certain differences in their arguments, it should be possible to disregard these differences.

For instance, in module `Sets-of-Naturals` the two different terms `add(1, add(0, emptyset))` and `add(0, add(1, emptyset))` obviously are intended to represent the set $\{0, 1\}$. So for any value `x` the terms `is-in(x, add(1, add(0, emptyset)))` and its variant `is-in(x, add(0, add(1, emptyset)))` will give the same answer. Hence we cannot distinguish between these terms using only function `is-in`. If these two terms are internally modeled by the same object they are the same, so we may have introduced so-called *confusion* between two textually distinct terms which cannot be proven equal via the rules provided in the specification. On the other hand we may be able to provide more efficient implementations using this knowledge.

So we need a way to express private or *hidden* parts of the specification. In an algebraic specification this can be achieved through the introduction of hidden sorts and functions.

Term rewriting

An effective recipe for implementation is wanted. This cannot be given in general since it depends on the semantics used. Most popular at the moment is the Term Rewriting approach, wherein all equations are replaced by rewrite rules, by giving a direction to the equations.

For instance, the module `Naturals` contains among others the following equations:

```
[n1] plus(0,n)          = n
[n2] plus(succ(m),n)    = succ(plus(m,n))
```

which may be systematically (from left to right) replaced by the rewrite rules:

```
[n1] plus(0,n)          -> n
[n2] plus(succ(m),n)    -> succ(plus(m,n))
```

meaning that `plus(0,n)` must be rewritten to the more simple form `n`, and `plus( succ(m), n)` to `succ( plus(m, n))`. So (proof by structural induction) any term containing `plus` will be reduced to one not containing this function after repeated application of the rewrite rules.

However, for a really efficient implementation too much depends on the accidental configuration of the rules to expect much from this direct translation. E.g., if equation n1 had been written backwards instead (`n = plus(0,n)`), a direct implementation from left to right in a rewrite rule format might result in an expansion of the term `n` to `plus(0,n)` to `plus(0,plus(0,n))`, etc., *ad infinitum*. This can be successfullly solved again in many ways, e.g., via Knuth-Bendix completion. For an overview, see [COMPASS91].

Hence implementation via program transformations seems to be the way to better implementations. Chapter 4 gives one approach to make this possible.

1.9 Disadvantages of formal specifications

Because this thesis aims at showing the advantages of formal specification, to balance the arguments provided it is also important to look at the problems with formal specifications. The wary specification writer will want to know about these disadvantages to be able to avoid them.

- They require a degree of *training* in formal methods. Some formalisms require more practice than others, but all do. The widespread lack of training can cause problems in two ways.
  - Information can be lost in the formalization process due to the fact that the specifier did not know how to describe this information.
  - The specification can be beyond the level of understanding of the original specifier, so validation of the formal specification becomes a serious problem.
- Even in cases where the semantics of the specification are clearly fixed from a mathematical point of view, they should also *agree with the* intuitive *understanding* of all the possible models by the writer of the specification. Otherwise non-standard models (of which the specifier is unaware) may be regarded as the best way to implement (by the programmer). An example of a non-standard model may be found in section 1.8.
- The implementation is often restricted by details in the specification, hence it must be possible to distinguish between relevant and irrelevant detail. The latter is much helped if *declarative* specifications are possible, since these provide

little detail which can be construed as necessarily required for the implementation. E.g., the list structure is not essential in the `Sets-of-Naturals` module in Figure 1.3.

Considering the advantages provided by formal specifications it is felt that the trade-off between formal and informal specifications will result in a preference for the former. The educational effort needed for more wide-spread practical application is the most serious obstacle, and readable specifications can provide the necessary push.

Algebraic specifications take care of many of the problems with formal specifications mentioned above:

- the degree of formal training is kept to a minimum for people moderately familiar with mathematics,
- they can be quite readable with a good choice of names for functions, constants and sorts,
- at least the initial semantics are intuitively clear to the specifier, and
- declarative specifications are possible.

1.10 An overview of the thesis

This thesis revolves around the use of modular algebraic specifications in programming. Chapter 2 shows the problems encountered with informal specifications and the problems with the subsequent formalization process. Our end goal is a modular algebraic specification, and the choice of possible models together with the ambiguities in the original informal specification made this a non-trivial task.

The power of algebraic specifications is illustrated in Chapter 3, where the semantically most difficult part of imperative languages, the **goto**-statement, is specified in a relatively elegant way.

The core is then provided by Chapter 4, which links Modular Algebraic Specifications and efficient implementations of selected functions. This Chapter states exact, practical conditions for the inclusion of efficient implementations in importing programs. A link to Hoare Logic specifications for individual functions, a convenient starting point for program transformations as shown in [DR86], is provided.

Chapter 5 finally applies the techniques to the problem from Chapter 2 to come up with an average case efficient solution. The transformations are kept initially at the specification level, thus keeping a high level of transparency of the proceedings.

Results

This thesis shows the following:

- The process of formalization is difficult, even when clear semantics are made available. Chapter 2 illustrates with a real-world example (the Swiss system) how imprecise even precise statements in natural languages can be. Hence it is argued that automated support is needed for formal specifications.

- Modular algebraic specifications are a powerful and flexible specification mechanism. This is illustrated with two original examples:

  - the Swiss system in Chapter 2, and
  - the notoriously difficult programming language construct of the **goto** in Chapter 3.

- Chapter 4 provides new observational semantics of modular algebraic specifications which allow for separate implementation of modules. This opens possibilities towards:

  - transformational techniques for program derivation;
  - program reuse (see above).

  A small price has to be paid in terms of extra proof obligations, but it has been shown that those obligations can be reduced to simple checks in almost all practical cases.

- Chapter 5 shows the power of derivational techniques, especially at the specification level. This resulted in a new implementation of the Swiss system with on average linear behaviour, where application of standard algorithms for Combinatorics would result in a cubic algorithm.

Chapter 2

Formalizing Informal Requirements – Some Aspects

N.W.P. van Diepen
H.A. Partsch

Formal specifications are nowadays considered as an important intermediate stage in the software development process. There are various approaches for constructing an efficient program satisfying a given formal specification. The formalization process, however, has not yet been investigated as thoroughly. Thus, it is still one of the main sources for inconsistencies between the wishes of the customer and the program finally delivered. Some problems to be solved during formalization are identified and illustrated with a real-world example.

2.1 Introduction

In its widest sense, software development means

> *"given a problem, find a program (or a set of programs) that (efficiently) solves the problem"*

where program may be taken as synonymous with software.

The major difficulty in software development is caused by the fact that the original problem description usually consists of a bunch of half-baked wishes which are neither precise or detailed, nor even complete. The program, by nature, has to be precisely defined and fully detailed up to each single instruction. It is obvious that software development done in one large step to bridge the huge gap between these extreme positions is doomed to fail, i.e., the resulting software probably does not work as expected.

There are various reasons why software might not work properly. Very often, the problem given originally was simply misunderstood or misinterpreted. Therefore, it is widely accepted today that the process of software development should be broken into smaller, manageable, steps in the framework of so-called "life cycle models". A

minimum requirement is a decomposition into two steps (frequently called "requirements engineering" and "program construction") with a precise, possibly formal, statement of the problem as an intermediate stage (cf., e.g., also [BCG83], [Agr86], [BMPP89]).

Such a formal problem specification states precisely and unambiguously the "task" to be fulfilled by the software, i.e., it describes what the problem is without giving a direct solution or even the details about its implementation. Additionally, it entails a "separation of concerns" which allows early checks on whether the informal wishes are properly reflected and thus prevents superfluous implementation work.

There are various approaches focusing on the program construction part of this development paradigm, viz. how to construct an efficient program that satisfies a given formal specification, e.g., by transformations (for overviews, see [PS83], [Fea86]), or assertional techniques ([Dij76], [Gri81], [Bac86]). The requirements engineering part, although at least as important, has not yet as thoroughly been investigated in the context of these new approaches and paradigms. However, a lot of work in this area has been done within traditional software engineering. Therefore, in the following section we attempt to shed some light on the problems to be encountered from a somewhat wider viewpoint. In section 2.3 we then will touch upon the particular problems of formal specification, and in section 2.4 we introduce some ideas on how to obtain formal specifications from informal problem statements. Section 2.5 contains some concluding remarks.

2.2 Requirements specification

In traditional software engineering, a problem specification is usually called a *requirements specification.* It is defined as ([IEEE83]):

> *"A specification that sets forth the requirements for a system or system component; for example, a software configuration item. Typically included are functional requirements, interface requirements, design requirements, and development standards."*

where in turn *requirement* is defined by

1. *"A condition or capability needed by a user to solve a problem or achieve an objective."*

2. *"A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component."*

From practice it is known that requirements appear as a huge, unstructured and unreflected mass of information that has to be analysed, organized and documented in a suitable way. To deal with this mass, more has to be known about requirements and requirements specification, e.g.,

- what are the *contents* of the requirements specification, i.e., what different kinds of requirements can be found,
- which *general properties* should a formalism for requirements specification satisfy, and
- *how should one proceed* to obtain a (formal) requirements specification that properly reflects the original intentions.

These aspects will be dealt with in section 2.2.2 through 2.2.4. For illustration purposes, section 2.2.1 introduces a non-trivial example from practice, which will be used throughout this paper.

2.2.1 A practical example

The nature of informal specifications bears the risk of writing in broad generalizations without any technical depth. Hence we have chosen a real-world example, both to focus our treatment and to illustrate our views. This example, the so-called Swiss system, will be described in some detail in the remainder of this subsection. It has been selected because a real-world problem helps in keeping a fairly unbiased view of the subject. Furthermore, a good, for the purpose of human application sufficiently complete, not too lengthy, informal specification of this problem is known. Also, it is not one of the "insider problems" from computing science, which would cloud the discussion with various standard solutions.

The Swiss system

The Swiss system is a tournament system designed to allow many participants to play a chess tournament in a limited number of rounds, both avoiding the drawbacks of round-robin or "all play all" tournaments, familiar from most national soccer championships (i.e., limited capacity or long duration), and knock-out tournaments, known from tennis (i.e., fast dropouts). The system was introduced in 1895 by Dr. J. Muller in Zürich. Since then it has been used in many variations at chess tournaments, and (sometimes adapted to the circumstances) at bridge, dames and go tournaments as well. The basic idea is as follows:

1. in every round, each player is paired with an opponent with an equal score (or as nearly equal as possible);
2. two participants are paired at most once;
3. after a predetermined number of rounds the leading player wins.

So, in round one, some random pairing is made. In round two all the winners play each other. The same holds for the players with a draw, and the losers. If there is an odd number of winners one of them plays a person with a draw (or a loser, but only when there are no people with drawn games). In round 3, players with 2 points

play each other, players with $1\frac{1}{2}$ points, etc. Again, if we have an odd number of players then one is selected to play someone of an adjoining group.

Many variations of the Swiss system exist, mainly to acomodate for particular circumstances. For instance, in chess tournaments players heavily favour playing with the white pieces. Hence colour allocation is important to ensure fair competition. Or participants may not wish to play their own clubmates since they can do so at home.

Various attempts have been made to implement the pairing algorithm of the Swiss system. However, a really satisfactory solution is, to our knowledge, still non-existent. Van den Herik ([Her88]) recently reported on a partially unsuccessful attempt by some students, called ZORBA (for "Zwitsers Op Rating BAsis" — Swiss on rating base). Rather than being forced to keep up with the original problem specification, they were able to influence the description of their version of the Swiss system during implementation. This has been done to allow for an easier implementation on the one hand, and to eliminate ambiguities in the description discovered during the implementation effort on the other. The effort of finding a better description even resulted in a new version of a rulebook for the Swiss system, ([GH88]). Still, even under these rather optimal implementation conditions, some problems remained in the final version, mainly concerning the problem of finding pairings in extreme cases.

Problems similar to the Swiss system pairing problem have been studied in combinatorics (cf. [PTW83]), e.g., the problem of allocating people to jobs in the most efficient way. Unfortunately, for our problem, which can be seen as a generalization of the job allocation problem, there is no known efficient solution.

A rule set for the Swiss system

To focus our attention we have taken the description of the Ratings controlled Swiss system (U.S. Chess Federation form) from [Kaž80] pp. 31-39, in a condensed form, leaving out some variants and exceptions. The essential rules for this version are:

0. *Initial remarks.* This Swiss system version assumes that all participants are given a rating, describing their playing strength. Unrated players are given a guessed rating, so it is assumed that each player is rated before the tournament, with ties decided at random. This *rating order* remains the same throughout the tournament and is used heavily in making pairings.

1. *Pairing cards.* A pairing card is made out for each player on which the tournament director records for each game the colour of the player's pieces, the opponent's name and identification number, the player's score in the game, and the player's cumulative tournament score.

2. *Identification Numbers.* After the entry list is closed, all pairing cards are arranged in the order of the players' ratings. Players with identical ratings are arranged by lot. Then the identification numbers of all players are entered on the pairing cards, starting with the highest-rated player as No. 1.

3. *Byes.* If the total number of players in any round of a tournament is uneven, one player is given a bye. A player must not be given a bye more than once. In the first round the bye is given to the player with the lowest official rating, in subsequent rounds to the lowest-ranked eligible player, rank in this case being determined first by score, then by official rating.

4. *Scoring.* The scoring is one point for a win or bye, one-half point for a draw, zero for a loss.

5. *Basic Swiss system Laws.* All pairings are subject to the following basic Swiss system Laws.

   (a) A player must not be paired with any other player more than once.

   (b) Players with equal scores must be paired if it is possible to do so.

   (c) If it is impossible to pair all players with equal scores, every player who is not paired with an opponent with equal score must be paired with an opponent whose score is as close to his own as possible.

6. *Pairing the first round.* After the bye, if any, is given, the pairing cards are arranged in rating order and are divided into two equal groups. The players in the top half are paired in consecutive order with those in the bottom half. For example, if there are forty players, No. 1 is paired with No. 21, No. 2 with No. 22, etc.

7. *Pairing subsequent rounds – score groups and rank.* In these rules the term 'score group', or simply 'group', is used in reference to a group of players having the same score. Sometimes a group may consist of only one player. Individual 'rank' is determined first by score, then by rating order.

8. *Order of pairing groups.* In general, the order of pairing is from the group with the highest score down to the group with the lowest score. Occasionally the pairing of the lower score groups may have to be adjusted to conform to the basic Swiss system Laws, if many of the players in those groups have met before.

9. *Method of pairing each score group.* In the second and as many of the subsequent rounds as possible, the players are paired as follows:

   (a) Any odd men are paired first as described in rules 10-12.

   (b) Within each score group, after the odd man, if any, has been removed, the cards of the remaining players are arranged in rating order and divided into two equal sections. The players in the top half (with the higher ratings) are paired with those in the bottom half (with the lower ratings) in as close to consecutive order as possible. Transpositions in the bottom half of a group are made to make the pairing conform to the basic Swiss system Laws and to give as many players as possible their due colours

(rules 15-17). If it is impossible to meet the two requirements just mentioned, one or two players in the top half may be interchanged with one or two players in the bottom half. Every effort should be made, however, to observe the principle of pairing the higher-rated against the lower-rated players.

Note. Directors differ somewhat in their exact methods for implementing this procedure, but any reasonable method, followed consistently, is acceptable.

10. *Odd men.* If there is an odd number of players in a score group, the lowest-ranked player is ordinarily treated as the odd man. However, the pairings in the group must accord with the basic Swiss system Laws. Sometimes two players who have met in a previous round must be treated as odd men because there is no possible way in which either of them can be paired in their original group.

11. *Method of pairing one odd man.* The odd man is paired with the highest-ranked player he has not met in the next-lower score group.

12. *Method of pairing more than one odd man.* If there are two odd men to be paired, the order in which they are paired is determined by their rank according to rule 7. If both cannot be paired, rank determines which is paired and which is moved to another group.

13. *Colour allocation – general principles.* The primary objective is to give white and black (nearly) the same number of times to as many players as possible. After the first round, as many players as possible should be given their due colours (rules 15-17).

14. *First round colours.* In the first round the colour assigned to all the odd-numbered players in the top half is chosen by lot, and the opposite colour is given to all the even-numbered players in the top half. Opposite colours are assigned to the opponents in the bottom half of the field as the pairings are made. (Once the first round colours are thus chosen by lot, rules 15-17 preserve equitable colour allocation, and no further lots are necessary.)

15. *Due colours in succeeding rounds.* As many players as possible are given their due colours as described in rules 16 and 17, so long as the pairings conform to the basic Swiss system Laws. Equalization of colours takes priority over alternation of colours.

16. *Equalization of colours.* As many players as possible are given the colour that equalizes the number of times they have played with the white and black pieces. When it is necessary to pair any two players who are due to being given the same equalizing colour, the higher-ranked player has priority in getting the equalizing colour, whether white or black.

17. *Alternation of colours.* After colours have been equalized in a round, as many players as possible should be given, in the next round, the colour each received in the first round of the tournament, the purpose being to continue alternation of colours. When it is necessary to pair any two players who are due to be given the same alternating colour, the higher-ranked player has priority in getting the alternating colour, whether white or black. However, if another pairing can be made in accordance with the basic Swiss system Laws, a player should not be assigned the same colour three times in a row. Interchanges between the top and bottom halves should not be made simply to preserve alternation of colours.

Further constraints

Some additional constraints are in order for a Swiss system program to be used in practice. Usually tournaments are held at a place where the computing facilities are limited to one microcomputer. Also time between rounds is typically very limited, with the need to process all results, including the pairing, in less than 30 minutes. Preferably, no special skills should be required from the user, both at the level of application of the Swiss system, and application of the program. So without going into much detail we would like to fix the following environment conditions:

1. the program should run on commonly used microcomputers without assuming a particular configuration or special capabilities of the device (e.g., size of the screen);

2. the program should be able to make a pairing within 5 minutes for groups of n players, $12 \leq n \leq 1000$, for rounds up to $\lfloor\sqrt{2n}\rfloor$;

3. the program should be usable by someone with only basic knowledge of the use of a computer and the Swiss system; and

4. the produced pairings should be of good quality, i.e. a manual check is not likely to significantly improve upon it. (Note: this is not a trivial requirement. ZORBA has been used as a shadow system to manual pairings.)

In the sequel these constraints will be referred to as constraints no. 1 through 4.

2.2.2 Different Kinds of Requirements

The Swiss system example shows very clearly that in practice, requirements may be of varying nature and quality. Therefore in this subsection we try to elaborate on the various kinds of requirements to be expected in a real-world problem.

Roughly, requirements can be split up into functional requirements and non-functional ones (cf. [Yeh82]). More detailed and comprehensive characterizations can be found, e.g., in [Rom85] and [KPR87].

Functional requirements deal with the behaviour of a system and its environment ("conceptual model", [BG79]). Typically they comprise of:

- inputs to the system and their constraints (e.g., the data on the "player cards", cf. rule 1, or the scoring of results, cf. rule 4),
- functions the system is able to perform (e.g., making a new pairing),
- outputs and other reactions of the system (e.g., updating and printing the pairing produced, asking for help in ambiguous situations);

The focus of the remainder of the paper will be on functional requirements, so we will not go into further detail here. Rather, we treat non-functional requirements in this section to a broader extent, to be able to dispense with more than casual reference furtheron.

Non-functional requirements (sometimes called "constraints") can be divided into different categories:

(a) quality attributes of the desired individual functions:

- performance (time, storage, workload, throughput – cf. constraint 2),
- maintainability (changes in individual functions should be feasible in a local way),
- reliability (failure safety (e.g., a system crash should not destroy all previous input), robustness, integrity of internal information, error-recognition, error-handling and survivability – cf. constraints 2 and 4),
- portability (cf. constraint 1),
- adaptability (cf. constraint 1, e.g., a possibility for adapting to exploit a full-size screen),
- compatibility with existing systems (e.g., a parser, a file system, a database of game results),
- reusability (i.e., modules should be structured, parameterized, and documented in a proper way),
- flexibility and extensibility (in order to satisfy additional requirements during the system lifetime),
- traceability (i.e., the possibility of recognizing the relationship between the original requirements and the specified functions),
- user comfort (cf. constraint 3);

(b) requirements for the implementation of the system:

- tools or devices to be used (e.g., existing software/hardware – cf. constraint 1, minimum memory requirements),
- interfaces with already existing components (e.g., with a text formatting system to produce camera ready forms for the tournament results),
- use of existing tools (programming language, operating system, hardware),

- documentation (describing, e.g., details about how to install the system);

(c) requirements for the development process:

- global development strategies (a division of the system into "independent" components),
- methods, languages, tools to be used (often some not explicitly mentioned standard known to all programmers involved),
- available resources (manpower, budget, deadlines – cf. constraint 1),
- quality attributes to be achieved and standards to be obeyed (cf. constraint 4);

(d) requirements for test, installation and maintenance:

- physical constraints (size, weight),
- availability of qualified personnel,
- skill level considerations (cf. constraints 3 and 4),
- spatial distribution of components (e.g., availability of a nearby printer);

(e) economical and political constraints:

- market considerations (cf. constraint 1),
- cost/benefit ratio (e.g., the trade-off between a general purpose and a customized system),
- legal restrictions (use of copyrighted software).

Informal requirements clearly cannot uniquely be characterized. This may cause trouble in practice because usually the different categories of requirements may be of greater or lesser importance to the user. Often requirements will be contradictory (e.g., fast, but on a small machine), which might not be clear at first inspection. Hence, finding these potential conflicts and stating a trade-off of relative importance between the requirements or defining primitives involved is important.

Obviously, there is a fundamental difference between functional and non-functional requirements: in order to be able to formulate non-functional requirements in a precise way, the functionality of a system has to be known. It is particularly for this reason that most of the approaches in requirements engineering mainly concentrate on providing formalisms to describe functional requirements. We form no exception here in concentrating on functional requirements for the remainder of this paper.

2.2.3 Desirable Properties of Formalisms for Requirements Definitions

Apart from being suited for expressing the various kinds of requirements as discussed in the previous subsection, a formalism for describing requirements has to cope with additional aspects originating in practical considerations, e.g., it has to be able to

deal with problems to be encountered in building a requirements definition and in constructing software that satisfies the requirements.

Typical problems when building a requirements definition are:

- uncertainty in what the problem is, due to uncertainty or to not measurable requirements (e.g., "if possible...", cf. rule 5b);
- incomplete information about the problem (e.g., the note to rule 9b);
- coordination and consistent integration of different sources of information (customer, user, technical expert, chess player, organizer, software developer);
- mass of information, easily leading to redundancy (e.g., rule 6 is a special case of rule 9b — before round 1 only one score group exists), and hence risking overspecification and inconsistency;
- different levels of detail in the requirements (e.g., rule 13, describing the general principle of colour allocation, versus rules 15-17, which give a procedure to be followed to implement this same general principle);
- exclusion of feasible solutions, (e.g., an additional rule in ZORBA excluded solutions with colour allocation more than 2 games out of balance, causing problems in special cases);

Typical problems in connection with the development process are:

- organization of the software development process in a manageable and reliable way;
- traceability and verification of an implemented system (with respect to its requirements);
- modification, enhancement, and maintenance;
- diversity of problems, making expertise gained in previous projects of uncertain value;
- estimation of the amount of effort needed.

In order to cope with these problems, the following properties of formalisms for requirements definition are desirable (cf. also, e.g., [BG79], [Fai85], [Hen81], [Rom85], [YZ80]):

precision and formality. In order to discover flaws at the earliest possible stage of the software development process, a precise and unambiguous statement of the problem to be solved is mandatory. The (implicit) desire for completeness and consistency can only be satisfied by a sufficient level of formality. Formality is also needed for being able to establish a formal correspondence (verification) between requirements definition and implemented system.

abstraction and structuring. Mastering complexity resulting from a mass of information requires abstraction mechanisms and suitable concepts for structuring.

conceptual integrity. A specification formalism has to be an integrated part of an overall software development methodology (cf. [Hen81]). Otherwise a smooth, manageable, and consistent development process leading to reliable software cannot be expected.

readability and understandability. A requirements specification is the interface (the "contract", [Bau81]) between client and software developer, and a means of communication among clients, users, experts, analysts, and designers. Thus, a formalism should be such that all parties involved will be able to read and understand the formulated requirements with reasonable effort.

modifiability. Software products are subjected to continuous changes (*"pressure of change is built-in"* [Leh80]), due to changing environments (and hence requirements). Obviously this entails a need for easy and consistent modifiability of a requirements definition, and a suitable formalism has to take care of this issue.

liberality. A specification should not enforce a single or a particular solution, but rather allow a variety of implementations (*"specification freedom"* [LF82], *"a family of solutions"* [YZ80]). Hence, the formalism should provide constructs allowing one to express that kind of freedom.

adequacy. A specification method has to provide means to increase confidence — especially on the customer's side — that the formal description really reflects his original intentions. For any kind of questions concerning the problem it should be possible to get answers that are formally justified on the basis of the specification.

wide range of applicability. Using a new formalism for every new problem is not feasible in practice. Therefore, a formalism for requirements definition must be capable of dealing with a wide range of different problems.

support by appropriate tools. Documentation, administration and analysis of the information contained in the requirements definition of a large problem are impossible to be managed without suitable tools. Thus, computer support for any kind of formalism should be aimed at. This also implies that a formalism should be machine supportable, e.g., by being unambiguously parseable.

2.2.4 How to Proceed

Assuming the availability of an adequate specification formalism, there is still the problem of methodology, i.e. how to proceed in order to build a requirements specification. A rough guideline is given in [RO85]:

> *"requirements engineering is a systematic approach to the development of requirements through an iterated process of analysing the problem, documenting the resulting requirements insights, and checking the accuracy of the understanding so gained."*

Individual activities that are to take place during this iterated process are, e.g.,

investigation of requirements:

- identification of the functional requirements in a dialogue between specifier, customer, and user,
- agreement on quality attributes (maybe including priorities or preferences) and other constraints,
- exploration of the environment for the system and its development;

formulation of requirements:

- precise formulation of all individual requirements,
- description of possible relationships between them,
- systematic structuring and classification;

analysis of requirements:

- formal checks for consistency and completeness,
- adequacy of the formulation,
- investigation of the technical feasibility ("Is the problem solvable at all by an algorithm?", "Are the requirements satisfiable with respect to the constraints on the intended environment?"),
- study on the economical feasibility (overall costs, schedule, required personnel, cost/benefit ratio, risks),
- rapid prototyping and other simulations (to test user acceptance).

We will come back to these issues when dealing with the formalization process in section 2.4.

2.3 Formal Specification

As already mentioned, formality is entailed by the demand of a requirements specification being consistent and complete. Furthermore, when asking for formality, there seems to be a consensus in the relevant literature that the level of formality provided by existing programming languages is not the appropriate one. For a requirements specification, the goal is a clear and precise description of the problem, rather than a formulation of a way to solve it.

Formality is a delicate issue, in particular since it cannot be seen independently of other desirable properties such as readability and understandability. On the one

hand one would like to have the precision and formal foundation of mathematics, but, on the other hand, one would prefer to have the understandability and wide range of applicability as provided by natural language (cf. [Hen81]).

Trying to achieve a compromise, traditional approaches (cf., e.g., [IEEE77], [Rom85]) introduce formal concepts only to an extent that is still manageable by a non-expert user. They provide only simple linguistic means for formulating the different kinds of requirements, mainly relying on an intuitive understanding of the semantics. Additionally, some of them are backed by methodological principles to ensure a systematic conversion of an informal problem statement into the respective formalisms. Nearly all of them, however, do not take subsequent steps in software development into account, i.e. they leave open how to obtain programs that solve the specified problem, and, furthermore, how to verify that these programs indeed meet the specification. Thus the essential drawbacks of these approaches are

- semantic imprecision (remaining ambiguities, no formal checks on consistency and completeness),
- lack of an integrated methodology (no formal verification),
- insufficient support for checking adequacy (no formally derived answers to questions on the problem).

There are a number of new approaches that focus on formalisms and integrated methodological support for (formally) constructing programs from a given formal specification of the problem. All of them assume a rigorous formal basis for an initial problem specification which is, e.g.,

- relational (e.g., Gist [Bal81], ERAE [DHR88]),
- functional (e.g., [Hen80], [BW88], VDM [Jon80]),
- predicative (e.g., [HGM86], [Bro87], Z [ASM79]),
- assertional (e.g., [Dij76], [Gri81], [Bac86]), or
- algebraic (e.g., ACT ONE [EFH83], ACT TWO [Fey86], ASF [BHK89], ASL [Wir83], CLEAR [BG80], COLD [JKR86], [FJOKRR87], LARCH [GH83], OBJ [GT77], [GM81], [FGJM85], PLUSS [Gau85], RAP-2 [Hußß87]).

Since these approaches do have a formal semantic basis, most of the above-mentioned drawbacks can be removed. This is at the price, however, of restricted expressiveness, new difficulties caused by the formalization process, and difficulties in reading and understanding.

Each of the approaches mentioned above has its strengths for particular aspects of a requirements specification. But none of them alone is powerful enough to cope with all kinds of requirements mentioned in section 2.2.2. Therefore, combinations and extensions, or even completely new formalisms, have to be looked for. How such an adequate formalism might look like is still a topic of research.

We are still convinced that an algebraically based approach is appropriate (cf. [Par86], [Par89]), because it meets nearly all the additional properties in connection with requirements definitions given in section 2.2.3 (cf. [Par87]). However, clearly, extensions are needed to enhance expressiveness, such as higher-order functions (cf. [Möl87], [MTW88]), specification-building operations (like those as, e.g., in [Wir83]) and relations (to be able to formulate certain non-functional requirements). Furthermore, in order to be able to formulate expressions over algebraic specifications or to express other non-functional requirements, conventional applicative constructs are needed, as well as more advanced concepts, such as non-determinism (for delayed design decisions w.r.t. specification freedom), predicate logic (for all kinds of conditions, properties, and constraints), modal and temporal logic (for real-time and other behavioral aspects), or traces (for parallel and distributed systems). Experiments with these and similar kinds of extensions are on the way.

For solving the problem of formalization, almost the same difficulties as in traditional requirements engineering have to be faced. Therefore, we suggest an approach to formalization which basically builds on experiences gained there, but also takes our envisaged enhanced version of an algebraic specification formalism into account. This will be the topic of the following section.

As to the matter of reading and understanding, attempts to provide understandability through translation of formal specifications into informal representation such as natural language text (cf. [Bau81], [Swa82], [Ehl85]), or graphics, seem to be promising, allowing even inspection by people without formal training.

2.4 The Process of Formalization

Formalization is the process in which an informally given problem is turned into a formal problem specification. As mentioned earlier (cf. section 2.2.4), this process generally comprises at least three essential sub-activities, viz.

- identification of the problem,
- formal description of the problem, and
- analysis of the formal problem description.

In the following subsections we will focus on each of these subtasks in turn. Some of the aspects mentioned in section 2.2.4 will be worked out in more detail, with an emphasis on formal specification.

2.4.1 Problem Identification

Problem identification means finding out what the problem is. The difficulties here mainly originate in the ambiguities and sources of misunderstanding inherent to the communication of different people by means of some informal language. Usually, the person who states the problem is not the one who is to describe it formally; additionally, due to different educational and professional backgrounds, they do not

speak the same language. Therefore, problem identification involves a mapping from one universe of discourse onto another, and the essential activity in problem identification concentrates on characterizing the universe of discourse in finding this mapping.

Usually a problem statement (implicitly) assumes basic knowledge about its context, the *problem domain*. To get a correct evaluation of the problem it is essential to make these implicit assumptions explicit, i.e., to first identify the characteristics of the problem domain (cf. "domain theory" [SL89]). Having done so, further steps in finding the above-mentioned mapping are

- the choice of a concept to describe the problem domain, with a suitable representation, and
- the definition of the problem in terms of the concept.

Following [Web74] we will use the notion *concept* for *"an idea or thought, especially a generalized idea of a class of objects; abstract notion"*. Hence, a concept of a (given) problem domain is an abstract view of the problem domain, free from irrelevant details, but suited to reflect its essential characteristics.

As we are concentrating on software systems, we can further rule out arbitrary technical concepts, needed in integrated technology as, e.g., process control, and focus our attention onto concepts from mathematics.

In order to illustrate our notion of a (mathematical) concept, we consider our Swiss system example again. The problem domain here is, among others, comprised of basic entities such as players and games, and rounds, combining to form a tournament. Thus, in a simplified view, a tournament is a structure consisting of players and games connecting them. One straightforward concept for this structure is an undirected finite graph. A finite directed graph is also a plausible concept, wherein the direction of an edge might be used to encode the colour allocation in the game, e.g., by pointing from white to black. This example can be pursued further by taking the concept of an edge labeled graph. The label associated with each edge could be used to encode the round number or the result of the game encoded by this edge.

Further examples of mathematical concepts are:

- sets, relations, mappings, functions (a round may be considered as a set of pairs of players; the pairing cards as a mapping of the players to various information associated with them),
- orderings and lattice structures (e.g., in the Swiss system example the set of players may be given a partial ordering according to their rating),
- algebraic structures (e.g., groups, rings, fields, sequences, bags, trees),
- relational structures (e.g., different kinds of graphs, Petri nets),
- formal systems (e.g., equational systems, grammars, automata, rewrite systems, deduction systems, systems of concurrent processes),

- differential equations, but also
- stochastic models, or
- topological and geometric structures.

The choice of a suitable concept already entails a tremendous gain with respect to precision, as the possibilities for misunderstandings and misinterpretations are restricted. Frequently, in addition, the choice of a concept even amounts to a solution of the problem, as certain tasks for certain concepts are already generally formalized or solved. Examples of this kind are:

- minima, maxima, (topological) sorting, or totalization in orderings, e.g., rule 2 describes the totalization of the rating ordering,
- construction and modification of particular algebraic structures,
- paths, cycles, or closures in relational structures, e.g., looking at the undirected graph representation of a tournament, the pairing problem can be formulated as: to find a list of pairs containing all nodes (players) once, such that no pair is connected in the current graph (no previous game exists),
- fixed points or zero valued arguments for equational systems,
- languages generated by grammars or accepted by automata,
- confluence and Church-Rosser properties of rewrite systems,
- deadlock or starvation in systems of concurrent processes, or
- congruence, similarity, and translation for geometric objects.

There is a lot of freedom in *choosing a concept.* Only in rare cases a concept is obvious or straightforward, because of concrete hints that can be found in the informal problem description. In our example a hint is, e.g., provided in rule 1, where a "pairing card", containing all necessary information, is associated to every player.

However, generally no such hints are available. Therefore, the choice of an adequate concept requires *decisions* with far-reaching consequences. Thus, not only the level of abstraction and the complexity of the formalization of the problem are affected, but later solutions to the problem are also enormously influenced. As a consequence, choosing an adequate concept is to be considered an art that requires great care, intuition and experience.

In general, a concept consists of:

- objects associated with certain object classes, e.g., "pairing cards",
- operations on the object classes, e.g., scoring, and
- relations between objects and/or object classes, e.g., "games from previous rounds" forms a relation in the domain of "pairing cards".

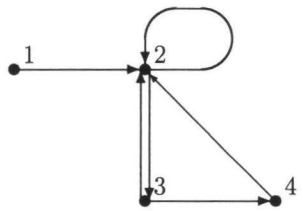
Since we did not assume any priorities among these constituents, this fairly general characterization of a concept comprises more restricted ones (to be found in various parts of the literature) that reflect certain "views" of a problem such as

- function oriented,
- data structure oriented,
- event oriented,
- control flow oriented, or
- data flow oriented.

E.g., in the Swiss system one could view the concept of making a pairing as a function from a list of rounds played and the player cards to a new round (function oriented). Another view is to consider a tournament as a tree of pairings (data structure oriented). A further point of view treats each player as a process in a concurrent system looking for a next pairing in case of a finished game (event oriented). These simple illustrations may give a rough impression of the problems concerned with choosing the right view.

In *representing a concept* one has to deal with a more detailed description of its constituents. Since there may be several representations of the same concept, again, a lot of freedom is provided here which involves further decisions.

The concept "finite directed graph", which we used in connection with our sample problem admits several (equivalent) descriptions. A general finite directed graph, such as:



can be defined as, e.g.,

(a) a set of nodes and a set of edges (represented by pairs of nodes):
$(\{1, 2, 3, 4\}, \{(1, 2), (2, 2), (2, 3), (3, 2), (3, 4), (4, 2)\})$;

(b) a set of nodes and a pair of incidence functions i and o which associate to each node the set of its predecessors and successors, resp.:

$$\begin{array}{llllll} (\{1, 2, 3, 4\}, & i\colon & 1 \to \emptyset & , o\colon & 1 \to \{2\} & \\ & & 2 \to \{1, 2, 3, 4\} & & 2 \to \{2,3\} & \\ & & 3 \to \{2\} & & 3 \to \{2,4\} & \\ & & 4 \to \{3\} & & 4 \to \{2\} &); \end{array}$$

(c) an adjacency matrix where component (i, j) has the value 1, if there is an edge from i to j, and 0 otherwise:

	1	2	3	4
1	0	1	0	0
2	0	1	1	0
3	0	1	0	1
4	0	1	0	0

Of course, the possibilities are not exhausted. However, it is obvious that the choice here could affect further developments in a significant way.

Having decided on a concept of the problem domain and a representation of the chosen concept, it remains to *define the problem in terms of the representation of the concept*, which, again, entails decision making.

If, for example, we decided on definition (b) above, we still would have to decide on the association of players and games with nodes and edges (the latter represented by incidence functions). One obvious possibility is to associate players with nodes, and games with edges. However, we also might associate both games and players with nodes, the former having two outgoing edges, one to white and one to black.

Which of several possible representations to choose, of course depends on further details of the problem to be solved. Thus, e.g., in the first association (i.e., players as nodes, games as edges), it is easy to check whether two players a and b have met before (in the terms of (b) above: either $a \in i(b)$ or $b \in i(a)$). However, a list of all games is difficult to produce. The second representation, on the other hand, gives easy access to individual games, but, for example, checking whether a and b have met is more difficult.

Other examples that illustrate the choice of possible concepts and their dependence on further details of the problem are:

- concept: *text*,

  representations:

  - sequence of characters (e.g., for a scanner),
  - sequence of words (e.g., for a parser),
  - sequence of sentences (e.g., for a translation program),
  - sequence of lines (e.g., for a line-oriented editor),
  - tree of chapters, sections, etc. (e.g., for the retrieval of indexed terms);

- concept: *mathematical formula*,

  representations:

  - string (e.g., for simple text processing),
  - tree (e.g., for evaluation or advanced typesetting).

In Section 2.2.2 we already commented on the distinguished role of functional requirements. This distinction becomes even more obvious with respect to formalization: the set of potential concepts is primarily determined by the functional requirements, whereas the choice among the members of this set, the choice of a representation of the selected member, and the choice on how to formally specify the problem in terms of the representation of the concept also takes non-functional requirements into account.

2.4.2 Problem Description

If a problem has been identified properly, its (formal) description amounts to translating the result of the identification process into constructs available in the formal specification language. In particular, this means

- mapping the representation of the concept of the problem domain onto available constructs, and
- giving an expression in the formal specification language that describes the task to be fulfilled in terms of the representation of the concept.

In the Swiss system example one needs a representation for the players, together with some of their characteristics, like "name" and "rating" (cf. rule 2). The representation of functions and relations like "rating order" of course depends on the choice of the representation. Possibilities for representation are, e.g., a set of players with functions for every property, or an array (list, set) of tuples, one for each player, containing all relevant characteristics. More formally, in the first representation one has a set of players P, and some functions like:

$$\begin{aligned} \mathit{rating}: &\ P \rightarrow \text{NATURAL} \\ \mathit{name}: &\ P \rightarrow \text{STRING of CHAR} \\ \mathit{order\_no}: &\ P \rightarrow \text{NATURAL}, \end{aligned}$$

while in the second representation one has an ARRAY of PLAYER p, where

$$\begin{aligned} \text{PLAYER} = \text{TUPLE}(&\ \mathit{name} : \text{STRING of CHAR} \\ &\ \mathit{rating} : \text{NATURAL} \\ &\ \vdots \\ ). & \end{aligned}$$

The definition of "rating order" now depends on the relevant entry in the tuple, or the relevant rating function, with some auxiliary entry or function to decide ties unambiguously. In a list or array representation these ties could be decided by the order in the list (array), or a new function or entry could be introduced as a totalization of the partial rating ordering. So in the first example one has a function

$$\mathit{higher\_rated} : P \times P \rightarrow \text{BOOLEAN},$$

defined by:

$$\begin{aligned} higher\_rated(a,b) = \; & rating(a) > rating(b) \vee \\ & (rating(a) = rating(b) \wedge \\ & order\_no(a) < order\_no(b)), \end{aligned}$$

while in the second example the function looks like:

higher_rated :
NATURAL × NATURAL × ARRAY of PLAYER → BOOLEAN,

now defined (disregarding border conditions on p) by:

$$\begin{aligned} higher\_rated(a,b,p) = \; & p[a].rating > p[b].rating \vee \\ & (p[a].rating = p[b].rating \; \wedge \; a < b), \end{aligned}$$

or even, if p is sorted on rating:

$$higher\_rated(a,b,p) = a < b.$$

Similarly, previously played games could be viewed as a graph, or as a list of pairs of players, or as a list of rounds, which in turn is a list of pairs of players, etc. A graph $G = (V, E)$ is described by a set of vertices V and a set of edges E. The graph representation might then look like (P, G) with P again the set of players and G the set of games between players from P. A game can be represented by a directed edge (a, b) for $a, b \in P$ ("a played with the white pieces versus b"). A function $have\_met : P \times P \rightarrow$ BOOLEAN is then easily defined as follows:

$$have\_met(a,b) = \exists g \in G : \; (g = (a,b) \vee g = (b,a)).$$

A more round oriented view of a tournament might contain a set of rounds $\{R_i : 1 \leq i \leq maxround\}$, wherein every $R_i \subset P \times P$. This allows the extension of function *have_met* with a round number to $have\_met : P \times P \times NATURAL \rightarrow$ BOOLEAN which could be defined as:

$$have\_met(a,b,r) = \exists i \in \{1 \ldots r\} : \; [\exists g \in R_i : \; (g = (a,b) \vee g = (b,a))],$$

or, in a recursive way, as:

$$\begin{aligned} have\_met(a,b,0) &= \text{FALSE} \\ have\_met(a,b,r) &= [\exists g \in R_r : \; (g = (a,b) \vee g = (b,a))] \vee \\ & \quad have\_met(a,b,r-1). \end{aligned}$$

The addition of these two predicates *have_met* allows the expression of the most important property of a valid new round, i.e., every player gets a new opponent. In our graph version this looks like (R being a set of edges):

$$\begin{aligned} valid(R) \equiv \; & \forall (x,y) \in R : \; \neg have\_met(x,y) \wedge \\ & |R| = \lfloor \tfrac{|P|}{2} \rfloor \wedge \\ & |P - \{x \in P : (\exists y \in P : \; ((x,y) \in R \vee (y,x) \in R))\}| \leq 1, \end{aligned}$$

or in words: no pair in R has met yet, R is half the size of the set of players P, and at most 1 player in P is not included in the pairing. The same can be expressed just as easily in the round version as follows:

$$\begin{aligned} valid(R_k) \equiv\ & \forall (x,y) \in R_k :\ \neg have\_met(x,y,k-1) \wedge \\ & |R_k| = \lfloor \tfrac{|P|}{2} \rfloor \wedge \\ & |P - \{x \in P : (\exists y \in P :\ ((x,y) \in R_k \vee (y,x) \in R_k))\}| \leq 1. \end{aligned}$$

Note that the latter version of *valid* allows us to check whether the previous rounds of the tournament have been entered correctly so far by defining a function *valid'* as follows:

$$valid'(R_k) = \forall i \in \{1 \ldots k\} :\ valid(R_i).$$

Of course, the graph representation does not provide such a check, since information on the round in which the game is played is lost. This could be solved by marking the edges with round numbers.

While pushing our straightforward formalization further, an ambiguity has been discovered in rule 5 (the Basic laws of the Swiss system). Rule 5b states that the number of players paired with a differently scoring player should be minimal and rule 5c that the difference between all scores in pairs should be minimal. If $d : P \times P \rightarrow$ RATIONAL gives the difference in score for each pair, both requirements can be included in the definition of the predicate *good*, which should be true for an optimal pairing:

$$\begin{aligned} good(R) \equiv\ & valid(R) \wedge \\ & \forall R' \subset P \times P :\ (valid(R') \rightarrow\ |\{(x,y) \in R : d(x,y) \neq 0\}| \leq \\ & \qquad\qquad\qquad\qquad\ |\{(x,y) \in R' : d(x,y) \neq 0\}|) \wedge \\ & \forall R' \subset P \times P :\ (valid(R') \rightarrow\ \textstyle\sum_{(x,y)\in R} d(x,y) \leq \\ & \qquad\qquad\qquad\qquad\ \textstyle\sum_{(x,y)\in R'} d(x,y)). \end{aligned}$$

Now, suppose at a certain stage the top score group has three players, a through c, and the next group two players, d and e. Say, c is selected as odd man and a plays b; c has already played d and e, so they play each other, and c plays someone, say f, two groups below. However, if b is selected as odd man a plays c, b plays d (or e) and e (or d) plays someone from the group containing f. Convention has it that the latter pairing is preferable, but the former has more players playing someone with the same score (rule 5b), while the pairing of c is the best possible according to rule 5c.

Similarly to other sub-activities of formalization, decisions are necessary here, too, depending on the particular specification language. Whereas translation of the representation of the concept into available language constructs in most cases will be straightforward, the formulation of the problem proper as an expression in the specification language usually again leaves a lot of freedom.

None of the decisions to be taken during the formalization process is unique, as we tried to illustrate by the simple examples above. Therefore a prime concern of any formalism for formal specification of problems is the provision of as much flexibility as possible in order to allow the adequate formulation of many possible

representations of a variety of different concepts. Ideally, there should be a one-to-one correspondence between concepts and constructs.

At least, however, any formalism for the formal specification of some task has to offer constructs that allow the representation of the constituents of a concept, i.e., objects and object classes, operations, and relations, and the formulation of expressions that reflect that task.

Conventional programming languages allow the definition of objects and object classes (called "modes" in ALGOL and "types" in Pascal), operations and relations (by means of function and procedure declarations), as well as arbitrary expressions. Therefore, programming languages are to be considered as specification languages, too.

However, traditional programming languages only allow the formulation of determinate, operational specifications. Likewise, new object classes can be defined only in a constructive, hence operational, way. Additionally, not all constructs offered by programming languages are really suited for problem specifications, as some of them, such as statements, procedures, loops, or pointers, are too implementation- or even machine-oriented. Hence, their use in problem specification would lead to too "low" a level of abstraction.

Consequently, a suitable specification language will contain only those constructs of traditional programming languages, such as function declarations or expressions, that are appropriate for formulating problem specifications on a rather "high" level of abstraction. Additionally, in order to overcome the above-mentioned restrictions to determinate, operational specifications, further constructs have to be provided for, like:

- formulating indeterminate specifications (e.g., it is not a good idea to replace the random allocation of the rating order (rule 0), by the order of entry, since this puts a premium on entering late because a weaker opponent is to be expected according to rule 6),
- expressing descriptive specifications (e.g., the note to rule 9b, which leaves to the tournament director some tricky and tedious but rather irrelevant details), and
- defining object classes in a non-operational way (e.g., the description of "pairing card" in rule 1).

2.4.3 Analysis of the Problem Description

A specification is called a *formal* specification, if it is formulated in a *formal* language, i.e., a language whose syntax and semantics are explicitly established prior to its use. Thus, obviously, formal specifications entail the usual problems of "formal correctness" to be encountered when using a formal language, viz. correctness with respect to syntax and context conditions, that have to be checked before starting semantic analysis or even program development.

The "meaning" of a formal specification is defined by the semantics of the specification language used. Usually this is a partial mapping from syntactic constructs

to (sets of) semantic values. On this basis additional practically important *semantic properties* of formal specifications can be introduced such as

- *defined* (also *consistent* or *satisfiable*)

  A formal specification is called defined if it has a "non-empty meaning", i.e., if there is at least one semantic value associated with the specified problem; otherwise it is called *undefined* (or *inconsistent*).

- *determinate*

  A formal specification is called determinate if there is at most one semantic value associated with to the specified problem; otherwise it is called *ambiguous.*

- *redundant*

  A formal specification is called redundant if there exists a semantically equivalent specification which is "simpler".

Except for simplicity, these properties can be formally checked on the basis of the semantics of the specification language. There are, however, additional properties that are not formally verifiable. These properties characterize the relationship between the meaning of the formal specification and the originally intended problem. Examples of such properties are:

- *adequate*

  A formal specification is called adequate if its meaning coincides exactly with the original problem.

- *overspecified*

  A formal specification is overspecified if its meaning comprises not all of the solutions to the original problem.

- *underspecified*

  A formal specification is underspecified if its meaning comprises all solutions to the original problem and additional ones. Thus, in particular, an ambiguous formal specification is underspecified if the original problem is uniquely sovable.

Obviously, these latter properties are not independent of each other: an adequate specification is neither over- nor underspecified, but inadequacy does not necessarily imply over- or underspecification.

It is important to be aware of the above-mentioned additional problems, and checking the respective properties of a formal specification is an essential part of the formalization process. The process of formalizing a problem may only be considered finished, when the formal specification is syntactically correct, and its adequacy with respect to the originally given problem is ensured. For practical reasons, an analysis with respect to redundancy seems worthwhile, too.

Obviously, there is a causal relationship between the expressiveness of a specification language and the amount of effort that is to be spent for ensuring the adequacy of a formal specification. The fewer constructs a language offers, the "longer", and thus the more "complex", expressions describing the problem will be. Consequently, the relationship between the formal specification and the originally given problem will be less obvious, and thus, more difficulties will be encountered when reasoning about adequacy.

Adequacy is the ultimate goal to be achieved. In order to reach it, an analysis with respect to the semantic properties seems worthwhile, because it gives valuable information. Thus, for example, recognizing a formal specification to be undefined usually indicates a defect in the formalization process rather than unsolvability of the originally given problem. Likewise, an indeterminate formal specification of a problem which is known to have a unique solution implies inadequacy. Also, an examination of the specification with respect to overspecification and underspecification provides valuable insight w.r.t. adequacy. Very often, underspecification can be removed by simply adding further conditions. Similarly, overspecification frequently can be eliminated by weakening certain restrictions. However, checking these properties is not sufficient. Further considerations with respect to adequacy are necessary, which, again, may lead to redoing (parts of) the formalization process.

2.4.4 Structuring

So far we did not pay any attention to the size of the problems to be specified. In fact, we even assumed that the formalization process as introduced in the previous subsections is not affected by problems in managing complexity mainly originating from the size of some task. In practice, however, size is a problem, and mastering the resulting complexity by introducing a suitable structure is an essential part of the formalization process. Also, the specification itself has to be built in a structured way.

In principle, there are two strategies for introducing structure: proceeding top-down or bottom-up.

Top-down proceeding is an iterated process that starts with the task as a whole and tries to split it up into smaller sub-tasks which in turn are subject to further decomposition. This process ends, if a suitable level of refinement is reached.

Technically, each step in a top-down proceeding consists of two alternating activities: "decomposition" and "elaboration".

Decomposition means "to break up or separate into basic components or parts" [Web74]. This includes identification of the parts, a clear statement on their respective interrelation, as well as the formulation of the original task in terms of the newly introduced components.

Elaboration means "to work out carefully; develop in great detail" [Web74]. Elaboration aims at providing meaning for the parts introduced in decomposition. This may be done by either referring to existing basic concepts or by initiating another decomposition step.

Within the framework of algebraic specification the combination of decomposi-

tion and elaboration just described amounts to introducing a new type. Decomposition roughly corresponds to introducing the signature of a type (i.e. the syntactic part) whereas elaboration aims at providing a semantics in the form of appropriate axioms for the object kinds and operations introduced by the preceding decomposition step.

Bottom-up proceeding is also an iterated process that starts from the details of a problem and aims at composing them into larger units (at a higher level of abstraction) until the level of the entire system is reached.

As with top-down proceeding, each step in bottom-up proceeding consists of two alternating sub-activities, viz. "composition" and "specialization".

Composition means "to put together; put in proper order or form" [Web74]. Composition comprises the introduction of new entities, as well as a precise statement on the components of this new entity and the way how they are to be combined in order to make up a whole.

Specialization means "1. to make special, specific, or particular; specify. 2. to direct toward or concentrate on a specific end" [Web74]. Usually, entities introduced by composition are too general for the particular task at hand. Specialization then tries to "adjust" these entities for the particular needs of the respective problem.

Within the framework of algebraic type specifications, bottom-up proceeding starts with a predefined collection of basic types (e.g., for numbers, truth values, characters, etc.) and basic type schemes (e.g., sequences, sets, bags, maps). Composition then means the definition of new types using suitable operations ("type constructors"). By specialization all those operations that are not needed are skipped from the list of visible constituents ("hiding"). Additionally, specialization can also introduce further restrictions on the operations and types ("constraints").

Both top-down and bottom-up proceeding as introduced above are idealistic views. In practice, both approaches will be used on the background of previous experience which always influences the proceeding in the opposite direction. Thus, for example, top-down proceeding is usually influenced by the availability of predefined types and type schemes or by certain ideas on the low level representation. Likewise, in bottom-up proceeding, composition, and in particular specialization, always will be done with the ultimate goal, viz. the entire system, in mind.

2.5 Conclusions

An attempt has been made to highlight some of the aspects and problems to be encountered in formalizing informal requirements. We favour writing the requirements document directly in a formal language, since it allows for early checking for completeness and consistency of a specification which is lacking in more informal methods. If a mistake can be detected early, the cost of repair is known to be relatively low. If detected later all kinds of followup from such a mistake have to be corrected too.

We know that it will be difficult to get acceptance of this view from people working in the field, since for practical requirements engineering, aspects such as

understandability or non-functional requirements have to be covered by formal approaches too. Therefore, further research in these directions has to be initiated.

The request from industry for a more rigorous approach is growing, but there is still a huge gap between ideas at universities on formal specification techniques and the day-to-day problems encountered in practice to be filled. To this end, further work is necessary in connection with:

- an *integrated methodology*, which provides sufficient guidelines for the practitioner,
- *software support*, e.g., in the form of tools to aid the process of formalization, or transformation systems for a safe transition from formal specifications to efficient programs, but above all
- *knowledge transfer*, in other to make all these beautiful ideas less academic and more usable for the practitioner.

Much can be learned by studying classical software engineering techniques, especially in the field of non-functional requirements. The results there should be applied to provide valuable information about necessary extensions of current formal specification methods, and about methodical guidance and software support needed to aid in practical application of formal specification methods.

Chapter 3

SMALL – dynamic semantics of a language with GOTOs

The algebraic specification of the semantics of SMALL – a programming language designed to illustrate specifications in denotational semantics – is given. Focus of attention is the specification of the semantics of **goto**-statements and the modular build-up of a language specification.

3.1 The specification of jump-statements

In Chapter 2 of [BHK89] the toy programming language PICO (the language of **while**-programs) is described in detail. PICO's small supply of language constructs leaves room for investigation in the specification of the semantics of more involved statement types.

The language to be specified in this chapter is SMALL, designed by Gordon [Gor79] as an example language to illustrate specifications in *denotational semantics*. SMALL is built in layers to allow one to concentrate on the difficulties of specifying a certain language construct while other constructs are excluded. In particular we are interested in the way **goto**s are defined in both formalisms: the denotational definition uses continuations (i.e. higher-order functions) for this purpose while our algebraic formalism is restricted to first-order functions. The freedom allowed by **goto**-statements makes it one of the most difficult classical programming primitives to specify. Hence specification of this construct is a serious test for any formalism. An earlier algebraic specification of **goto**-statements can be found in [BDMW81]. Their style of specification is similar to the specification in section 3.5 here. We are also interested in the question how to capture the various layers of SMALL in a single, modular, definition.

The next section describes the abstract syntax and (informally) the semantics of the SMALL kernel language (SMALL proper), followed by the syntax and semantics of an extension with **goto**-statements. An algebraic specification of the semantics of the kernel language is given in section 3.3, to provide a basis for section 3.4, a specification of the extension with **goto**s. However, the specification in section 3.3 is written with the extension in mind. In section 3.5 an alternative, more elegant,

specification of SMALL without **goto**s is given. This specification is not immediately suited for addition of **goto**-statements. A specification of the extended language circumventing this problem is given. Section 3.6 provides a description of an "ad hoc" implementation of the specification of section 3.3 and 3.4. Chapter 5 of [BHK89] and [Wal91] treat implementation in greater detail.

3.2 Syntax and informal semantics of SMALL

3.2.1 Abstract syntax and informal semantics

In the sequel the kernel language will be called SMALL1 and the extension with **goto**-statements SMALL2. These indications will also be used in the names of the modules of the specification.

Basic values, identifiers and operators

Some primitive notions are needed to give a basis to the operations of a programming language. Firstly, a module `Booleans`, like the one in Chapter 1, is needed. Further notions are treated abstractly and are grouped into one module: `SMALL1-Primitives`, containing the sorts `BASICVAL` (the basic values of the SMALL kernel language), `IDNT` (identifiers) and `BINOP` (binary operators), together with an equality function on `IDNT` yielding a boolean.

Abstract syntax and informal semantics of SMALL1

The constructor functions for the abstract syntax are combined in a module named `SMALL1-Abs-Synt`. The sorts `DECL` (declarations), `DECLS` (lists of declarations), `EXPR` (expressions), `CMND` (commands), `CMNDS` (lists of commands) and `PROGRAM` (SMALL1 programs) are defined here.

Below follows a list of the defined constructor functions. A correspondence between these functions (and those of the next section) and the concrete syntax is given in section 3.2.2.

- the *program* constructor:

  ```
  program:  CMNDS -> PROGRAM
  ```

  This function corresponds to the root of the abstract syntax tree of a SMALL1 program. It turns a series of commands into a program.

- *command* constructors:

  ```
  abs-assign  : EXPR # EXPR           -> CMND
  abs-output  : EXPR                  -> CMND
  abs-proccall: EXPR # EXPR           -> CMND
  abs-if      : EXPR # CMNDS # CMNDS  -> CMND
  abs-while   : EXPR # CMNDS          -> CMND
  abs-block   : DECLS # CMNDS         -> CMND
  abs-ser     : CMND # CMNDS          -> CMNDS
  abs-skip    :                       -> CMNDS
  ```

  SMALL1 has rather conventional commands. Unusual features include the left-hand side of an assignation command, which is an expression that yields an identifier. Similarly, the first expression of a procedure call gives its name,

the second one gives the value of the (single) parameter. Every procedure has exactly one parameter.

A block consists of a list of declarations and a list of commands. Sequential composition of commands is modeled as a list with `abs-skip` as terminator.

- *expression* constructors:

```
absexp-basicval: BASICVAL             -> EXPR
absexp-read    :                      -> EXPR
absexp-ident   : IDNT                 -> EXPR
absexp-funcall : EXPR # EXPR          -> EXPR
absexp-ifexp   : EXPR # EXPR # EXPR   -> EXPR
absexp-binop   : BINOP # EXPR # EXPR  -> EXPR
```

A function call consists — like a procedure call — of an expression yielding its name, and again exactly one parameter. Since basic values are treated abstractly, no concrete binary operators have been defined and their appearance here is purely *pro forma*.

- *declaration* constructors:

```
absdecl-const: IDNT # EXPR          -> DECL
absdecl-var  : IDNT # EXPR          -> DECL
absdecl-proc : IDNT # IDNT # CMNDS  -> DECL
absdecl-fun  : IDNT # IDNT # EXPR   -> DECL
absdecl-ser  : DECL # DECLS         -> DECLS
absdecl-skip :                      -> DECLS
```

In declarations of constants and variables, the first component yields the new name and the second component states the initialization value. The second identifier of function and procedure declarations is the name of the parameter. The structure of declarations is list-like.

Abstract syntax and informal semantics of SMALL2

To enrich the SMALL1 abstract syntax with **goto**s in section 3.4 a module with the name `SMALL2-Abs-Synt` is built on top of `SMALL1-Abs-Synt`. It contains three additional constructor functions:

```
abs-goto     : IDNT        -> CMND
abs-labldcmnd: IDNT # CMND -> CMND
absdecl-label: IDNT        -> DECL
```

A **goto**-statement jumps to the *last* label with the name `IDNT` in the block in which the label is declared. Jumps into an inner block or a procedure are illegal, jumps out of a procedure or an inner block are allowed. Jumps into the body of loops continue with the rest of the body followed by the whole loop and the rest of the program.

3.2.2 Concrete syntax

In the remainder of this Chapter no concrete syntax for SMALL is needed. It is included here to help the reader with the intuitive meaning of the abstract syntax presented in the previous section.

SMALL1 **abstract syntax versus concrete syntax**

abstract syntax	**concrete syntax**
program constructor	*program*
`program(cmnds)`	**program** <list-of-commands>
command constructors	*commands*
`abs-assign(expr1,expr2)`	<expression1> := <expression2>
`abs-output(expr)`	**output** <expression>
`abs-proccall(expr1,expr2)`	<expression1> (<expression2>)
`abs-if(expr,cmnds1,cmnds2)`	**if** <expression> **then** <list-of-commands1> **else** <list-of-commands2>
`abs-while(expr,cmnds)`	**while** <expression> **do** <list-of-commands>
`abs-block(decls,cmnds)`	**begin** <list-of-declarations> ; <list-of-commands> **end**
`abs-ser(cmnd,cmnds)`	<command> ; <list-of-commands>
`abs-skip`	<empty-list-of-commands>
expression constructors	*expressions*
`absexp-basicval(basicval)`	<basic-value>
`absexp-read`	**read**
`absexp-ident(idnt)`	<identifier>
`absexp-funcall(expr1,expr2)`	<expression1> (<expression2>)
`absexp-ifexp(expr,expr1,expr2)`	**if** <expression> **then** <expression1> **else** <expression2>
`absexp-binop(binop,expr1,expr2)`	<expression1> <binary-operator> <expression2> <expression2>

abstract syntax	**concrete syntax**
declaration constructors	*declarations*
`absdecl-const(idnt,expr)`	**const** <identifier> = <expression>
`absdecl-var(idnt,expr)`	**var** <identifier> = <expression>
`absdecl-proc(idnt1,idnt2,cmnds)`	**proc** <identifier1> (<identifier2>); <list-of-commands>
`absdecl-fun(idnt1,idnt2,expr)`	**fun** <identifier1> (<identifier2>); <expression>
`absdecl-ser(decl,decls)`	<declaration> ; <list-of-declarations>
`absdecl-skip`	<empty-list-of-declarations>

SMALL1 will be augmented with **goto**-commands in section 3.4. The syntax will be enlarged as follows:

SMALL2 **abstract syntax versus concrete syntax**

abstract syntax	**concrete syntax**
`abs-goto(idnt)`	**goto** <identifier>
`abs-labldcmnd(idnt,cmnd)`	<identifier> : <command>
`absdecl-label(idnt)`	**label** <identifier>

3.3 Algebraic semantics of the SMALL kernel

3.3.1 The environment

To manipulate entities necessary to describe the behaviour of a SMALL program, a storage mechanism is needed. The basis for this storage mechanism is the `TABLE`, which will be a modified version of the data type in Chapter 2. It has two parameters: `Names` and `Entries`. The functions `null-table` (generates an empty table), `table` (puts a fresh *name–entry* combination in a table), `tablech` (changes an entry corresponding to a given name) and `lookup` are given. An equality predicate must be defined on the names. These sorts and functions are bundled in module `Tables`.

Compared with the module `Tables` in the previous Chapter the function `table` is split here into an addition function (also called `table`) and a modification function called `tablech`. The reason for this will be explained below. The specification of `delete` on tables was not necessary for this Chapter.

```
module Tables
begin
parameters
  Names   begin sorts NAME
                functions eq: NAME # NAME -> BOOL
          end Names,
  Entries begin sorts ENTRY
                functions error-entry: -> ENTRY
          end Entries
exports
  begin
    sorts TABLE
    functions
      null-table:                       -> TABLE
      table     : NAME # ENTRY # TABLE -> TABLE
      tablech   : NAME # ENTRY # TABLE -> TABLE
      lookup    : NAME # TABLE         -> BOOL # ENTRY
  end
imports Booleans
variables name, name1, name2    : -> NAME
          entry, entry1, entry2 : -> ENTRY
          tbl                   : -> TABLE
equations
[  i] lookup(name, null-table) = <false, error-entry>
[ ii] lookup(name1, table(name2,entry,tbl))
         = if(eq(name1,name2),<true,entry>,
                              lookup(name1,tbl))
[iii] tablech(name1, entry1, table(name2,entry2,tbl))
         = if(eq(name1,name2),
              table(name1,entry1,tbl),
```

```
                table(name2,entry2,
                      tablech(name1,entry1,tbl)))
[ iv] tablech(name, entry, null-table) = null-table
end Tables
```

Both undeclared and multiply declared names are allowed. Looking up an undeclared name gives an error flag, looking up a multiply declared name returns the last entry only. This is done on purpose. The dynamic semantics specification in section 3.3.2 is supposed to operate on abstract syntax trees which have passed a static semantics check. Hence both undeclared and multiply declared names should not occur anymore when these tables are used for SMALL's dynamic semantics.

Here and in the remainder of this Chapter no attempt is made to present the most abstract version of a specification feasible. There are two reasons for this. Firstly, it is often much more convenient to specify some sort satisfying your needs than to specify exactly what your needs are by describing them. Secondly, intuition is helped with a more precise specification by hinting at (or providing) a certain model. The main drawback is the loss of generality, so one might lose too much freedom if one is careless.

SMALL has block structure (as in, e.g., Pascal). The elementary storage mechanism provided by `Tables` does not provide sufficient power to capture this structure in an easy way. Hence a new module `SMALL1-Tables` is built for this task on top of `Tables`. A function `blockmark` is introduced to separate blocks in a table. A function `removeblock` is also defined.

If a name does not occur in the topmost block either an addition must be made (when using `table` here), or the next block must be searched (when using `tablech`).

Sort `TABLE` is renamed to `SENV` (SMALL-environment). The sort `NAME` (in parameter `Names`) is bound to `IDNT` (from `SMALL1-Primitives`). The objects we want to put into the table have to be bound to sort `ENTRY` from `Entries`. Since this comprises objects of various sorts (e.g., declarations and basic values) an intermediate module `SMALL1-Env-Elt` is constructed to provide a common sort, called `ENVELT` (environment-element), and injection functions from the primitive sorts into this sort. Also, lists of elements of sort `BASICVAL` are added to sort `ENVELT` to model input and output in a rudimentary way, with operations `top` (returns the top element), `pop` (deletes the top element), `cat` (concatenates an element at the end) and `emptylist`. Because of its length this rather easy specification is left out. This intermediate sort is bound to `ENTRY`.

```
module SMALL1-Tables
begin
exports
  begin functions blockmark  : SENV -> SENV
                  removeblock: SENV -> SENV
  end
imports
  Tables
    { renamed by [TABLE      -> SENV,
```

```
                null-table -> null-senv]
    Names bound by [NAME -> IDNT,
                    eq   -> eq]
          to SMALL1-Primitives
    Entries bound by [ENTRY       -> ENVELT,
                      error-entry -> error-value]
          to SMALL1-Env-Elt
  }
variables idt : -> IDNT
          elt : -> ENVELT
          tbl : -> SENV
equations
[   v] removeblock(blockmark(tbl)) = tbl
[  vi] removeblock(table(idt,elt,tbl))
         = removeblock(tbl)
[ vii] lookup(idt,blockmark(tbl)) = lookup(idt,tbl)
[viii] tablech(idt,elt,blockmark(tbl))
         = blockmark(tablech(idt,elt,tbl))
end SMALL1-Tables
```

3.3.2 The dynamic semantics of SMALL1

The algebraic specification of the dynamic semantics of SMALL1 is straightforward.

Errors – and correspondingly error messages – are disregarded in this specification to avoid a longer and more cluttered definition. So the specification below only holds for correct programs in SMALL. The meaning of some incorrect programs on certain inputs is also defined, but purely as a side-effect.

In this specification the work is mainly carried out by evaluation functions for the elementary language constructs. Function **eval** is given either a program and input or a series of commands and an environment (containing both in- and output). **evalexpr** operates on an expression and an environment and **evaldecl** on a declaration or a series of declarations and an environment. The environment resulting from a correct evaluation contains the output and the (possibly exhausted) input.

An auxiliary constant **abs-blockend** is introduced to mark the end of the commands forming a block in the series of commands to be executed. The auxiliary function **cat** (short for con**cat**enate) is necessary to join series of commands. The reason for this rather clumsy style of specification becomes apparent when the specification is extended with **goto**-statements. Then it is a clear advantage that the remainder of the series (the second argument of **abs-series**) is available here. For an alternative style (close to the style in Chapter 2 of [BHK89]) the reader is referred to section 3.5.

```
module SMALL1
begin
exports
  begin
```

```
    functions
      eval          : PROGRAM # ENVELT -> SENV
      eval          : CMNDS # SENV     -> SENV
      evaldecl      : DECL # SENV      -> SENV
      evaldecl      : DECLS # SENV     -> SENV
      evalexpr      : EXPR # SENV      -> BASICVAL # SENV

      applyfun      : IDNT # BASICVAL # SENV
                        -> BASICVAL # SENV
      applybinop    : BINOP # BASICVAL # BASICVAL
                        -> BASICVAL

      abs-blockend :                   -> CMND
      cat           : CMNDS # CMNDS    -> CMNDS

      in            :                  -> IDNT
      out           :                  -> IDNT
    end
imports SMALL1-Abs-Synt, SMALL1-Tables
variables dcl                       : -> DECL
          dcls                      : -> DECLS
          exp, exp1, exp2           : -> EXPR
          cmd                       : -> CMND
          cmds, cmds1, cmds2        : -> CMNDS
          senv, senv1, senv2        : -> SENV
          bval, bval1, bval2        : -> BASICVAL
          idnt, idnt1, name, param  : -> IDNT
          oper                      : -> BINOP
          entry, input              : -> ENVELT
          bool                      : -> BOOL

equations
 [1] eval(program(cmds),input)
        = eval(cmds, table(out,emptylist,
                     table(in,input,null-senv)))
```

The constants out and in are the names of output and input.

```
 [2] eval(abs-ser(abs-assign(exp1,exp2),cmds),senv)
        = eval(cmds,tablech(idnt,envelt(bval),senv2))
        when <bval,senv1> = evalexpr(exp2,senv),
             <basicval(idnt),senv2> = evalexpr(exp1,senv1)
 [3] eval(abs-ser(abs-output(exp),cmds),senv)
        = eval(cmds,tablech(out,cat(entry,bval),senv1))
        when <bval,senv1> = evalexpr(exp,senv),
             <true,entry> = lookup(out,senv1)
```

niet te lezen


```
[4] eval(abs-ser(abs-proccall(exp1,exp2),cmds),senv)
      = eval(abs-ser(abs-block(
                        absdecl-ser(
                          absdecl-const(param,
                            absexp-basicval(bval)),
                          absdecl-skip),
                        cmds1),cmds),
             senv1)
      when <basicval(name),senv1> = evalexpr(exp1,senv),
           <true, envelt(absdecl-proc(name,param,cmds1))>
             = lookup(name,senv1),
           <bval,senv2> = evalexpr(exp2,senv1)
```

The procedure body is stored in the environment (by equation 19). Parameter binding is described by constructing a new block consisting of the declaration and initialization of the parameter followed by the procedure body itself. (Note that only call-by-value is modeled, and that only one parameter is allowed. Another extension to SMALL in [Gor79] treats function and procedure descriptions.)

```
[5] eval(abs-ser(abs-if(exp,cmds1,cmds2),cmds),senv)
      = if(bool,eval(cat(cmds1,cmds),senv1),
               eval(cat(cmds2,cmds),senv1))
      when <basicval(bool),senv1> = evalexpr(exp,senv)

[6] eval(abs-ser(abs-while(exp,cmds1),cmds),senv)
      = if(bool,
           eval(cat(cmds1,
                    abs-ser(abs-while(exp,cmds1),
                            cmds)),
                senv1),
           eval(cmds,senv1))
      when <basicval(bool),senv1> = evalexpr(exp,senv)
```

The expression must be of type boolean in the two equations above, otherwise no match can be found in the when-clauses.

```
[7] eval(abs-ser(abs-block(dcls,cmds1),cmds),senv)
      = eval(cat(cmds1,abs-ser(abs-blockend,cmds)),
                 evaldecl(dcls,blockmark(senv)))
[8] eval(abs-ser(abs-blockend,cmds),senv)
      = eval(cmds,removeblock(senv))
```

In equation 7 a block is created in the environment, and in equation 8 it is removed again.

```
[9] eval(abs-skip,senv) = senv
```

When there are no more commands to be executed the current environment contains the final result.

```
[10] evalexpr(absexp-basicval(bval),senv)
       = <bval,senv>
[11] evalexpr(absexp-read,senv)
          = <bval,tablech(in,pop(entry),senv)>
       when <true,entry> = lookup(in,senv),
            bval = top(entry)
```

The first element is taken from the list of input values (associated with the identifier `in`).

```
[12] evalexpr(absexp-ident(idnt),senv) = <bval,senv>
       when <true,envelt(bval)> = lookup(idnt,senv)
[13] evalexpr(absexp-funcall(exp1,exp2),senv)
       = applyfun(name,bval2,senv2)
       when <basicval(name),senv1> = evalexpr(exp1,senv),
            <bval2,senv2> = evalexpr(exp2,senv1)
[14] applyfun(name,bval,senv)
            = <bval1,removeblock(senv1)>
       when
         <true,envelt(absdecl-fun(name,param,exp))>
            = lookup(name,senv),
         <bval1,senv1>
            = evalexpr(exp,table(param,envelt(bval),
                                 blockmark(senv)))
```

A construction similar to the procedure call in equation 4 is used here to bind the parameter of a function call. The difference in treatment between procedure and function calls is a reflection of the asymmetry of the notions *command* and *expression* in SMALL1. Evaluation of an expression produces a value as result, while evaluation of a command is noticeable only by its side-effect on the environment.

```
[15] evalexpr(absexp-ifexp(exp,exp1,exp2),senv)
       = if(bool,evalexpr(exp1,senv1),
                 evalexpr(exp2,senv1))
       when <basicval(bool),senv1> = evalexpr(exp,senv)
[16] evalexpr(absexp-binop(oper,exp1,exp2),senv)
       = <applybinop(oper,bval1,bval2),senv2>
       when <bval2,senv2> = evalexpr(exp2,senv1),
            <bval1,senv1> = evalexpr(exp1,senv)
```

Function `applybinop` must be defined for every binary operator.

```
[17] evaldecl(absdecl-const(idnt,exp),senv)
       = table(idnt,envelt(bval),senv1)
       when <bval,senv1> = evalexpr(exp,senv)
```

No safeguard is given for constants, hence they could be assigned to. The check on the correctness of the static semantics should ensure that this will not happen.

```
[18] evaldecl(absdecl-var(idnt,exp),senv)
       = table(idnt,envelt(bval),senv1)
       when <bval,senv1> = evalexpr(exp,senv)
[19] evaldecl(absdecl-proc(name,param,cmds),senv)
       = table(name,
               envelt(absdecl-proc(name,param,cmds)),
               senv)
[20] evaldecl(absdecl-fun(name,param,exp),senv)
       = table(name,
               envelt(absdecl-fun(name,param,exp)),
               senv)
```

The bodies of procedures and functions are added to the environment.

```
[21] evaldecl(absdecl-ser(dcl,dcls),senv)
       = evaldecl(dcls,evaldecl(dcl,senv))
[22] evaldecl(absdecl-skip,senv) = senv
```

Here the definition of the dynamic semantics of SMALL1 ends. One auxiliary function still needs to be defined:

```
[23] cat(abs-ser(cmd,cmds1),cmds2)
       = abs-ser(cmd,cat(cmds1,cmds2))
[24] cat(abs-skip,cmds) = cmds
end SMALL1
```

3.4 SMALL with GOTOs

Module SMALL2 is defined by extending SMALL1 with the abstract syntax tree constructors introduced in SMALL2-Abs-Synt and by augmenting the evaluation functions and where appropriate the auxiliary functions to cope with these new functions.

The specification of module SMALL2, interspersed with commentary, takes the remainder of this section.

```
module SMALL2
begin
exports
  begin
    functions
      absdecl-lbldcmnd: CMNDS                -> DECL
      jmpcont         : IDNT # SENV          -> CMNDS # SENV
      continuation    : IDNT # SENV          -> BOOL # CMNDS
      search-cont     : IDNT # CMNDS         -> BOOL # CMNDS
      adjust-nesting  : IDNT # SENV # SENV   -> SENV
      saveprogram     : CMNDS # CMNDS # SENV -> SENV
      lookupprogram   : SENV                 -> CMNDS # CMNDS
      deleteprogram   : SENV                 -> SENV
      blockbody       :                      -> IDNT
      progrest        :                      -> IDNT
  end
imports SMALL1, SMALL2-Abs-Synt
variables dcls                       : -> DECLS
          exp, exp1, exp2            : -> EXPR
          cmd                        : -> CMND
          cmds, cmds1, cmds2, cmds3: -> CMNDS
          senv, senv1                : -> SENV
          idnt, idnt1, lbl           : -> IDNT
          bool, found, found2        : -> BOOL
          envlt                      : -> ENVELT
equations
[25] eval(abs-ser(abs-labldcmnd(lbl,cmd),cmds),senv)
       = eval(abs-ser(cmd,cmds),senv)
[26] eval(abs-ser(abs-goto(lbl),cmds),senv)
       = eval(cmds1,senv1)
       when <cmds1,senv1> = jmpcont(lbl,senv)
[27] eval(abs-ser(abs-block(dcls,cmds1),cmds),senv)
       = eval(cat(cmds1,abs-ser(abs-blockend,cmds)),
              deleteprogram(evaldecl(dcls,
                  saveprogram(cmds1,cmds,senv))))
```

Function evaldecl will need information about the program when declarations of labels are encountered. Hence the evaluation of a block has to be adapted. In

equation 27 the body of the block and the rest of the program are temporarily stored in the environment. These program fragments can be retrieved by function `lookupprogram` (specified in equation 29).

Equation 27 and equation 7 from module `SMALL1` both specify the evaluation of a block. When a block contains **label**-declarations equational derivations starting with equation 7 will not be able to get further on the first declaration of a label, while equation 27 will. When a block does not contain label-declarations, both equations together imply the equivalence of the two equations about `evaldecl` for such a block. Hence SMALL1 is implicitly a sublanguage of SMALL2. (If, accidentally, these equations lead to different environments for programs without labels, the specification would be incorrect, and probably even inconsistent.)

```
[28] saveprogram(cmds1,cmds,senv)
       = blockmark(
           table(blockbody,
             envelt(absdecl-lbldcmnd(cmds1)),
             table(progrest,
               envelt(absdecl-lbldcmnd(cmds)),
               blockmark(senv))))
[29] lookupprogram(senv) = <cmds1,cmds>
       when <true,envelt(absdecl-lbldcmnd(cmds1))>
              = lookup(blockbody,senv),
            <true,envelt(absdecl-lbldcmnd(cmds))>
              = lookup(progrest,senv)
[30] deleteprogram(table(idnt,envlt,senv))
       = table(idnt,envlt,deleteprogram(senv))
[31] deleteprogram(blockmark(senv))
       = blockmark(removeblock(senv))
[32] jmpcont(lbl,senv) = <cmds,senv1>
       when <true,envelt(absdecl-lbldcmnd(cmds))>
              = lookup(lbl,senv),
            senv1 = adjust-nesting(lbl,senv,senv)
[33] adjust-nesting(idnt,senv,
                    table(idnt1,envlt,senv1))
       = if(eq(idnt,idnt1),senv,
            adjust-nesting(idnt,senv,senv1))
[34] adjust-nesting(idnt,senv,blockmark(senv1))
       = adjust-nesting(idnt,senv1,senv1)
```

Some auxiliary functions will be used to describe the behaviour of the **goto**-construct. `jmpcont` selects from the environment the continuation of the program for a given label identifier. This function uses `adjust-nesting`, which deletes the part of the environment corresponding to inner blocks.

The most important functions are `continuation` and its auxiliary function `search-cont` which look for a continuation corresponding to a label at the moment it is declared. The first function selects the body of the block and the rest of

the program from the environment, and starts up the search for a continuation in the blockbody. When a continuation is found, the rest of the program is attached to this series of commands, preceded by an **abs-blockend**-marker.

```
[35] evaldecl(absdecl-label(lbl),senv)
       = table(lbl,envelt(absdecl-lbldcmnd(cmds)),senv)
       when <true,cmds> = continuation(lbl,senv)
[36] continuation(lbl,senv) = <bool,cat(cmds2,cmds)>
       when <cmds1,cmds> = lookupprogram(senv),
            <bool,cmds2> = search-cont(lbl,cmds1)
```

The scan of the blockbody is the task of function **search-cont**. Many statements are simply skipped (equations 37, 38, 39, 42 and 43). Equation 42 describes that it is impossible to jump into an inner block.

```
[37] search-cont(lbl,abs-ser(
                 abs-assign(exp1,exp2),cmds))
       = search-cont(lbl,cmds)
[38] search-cont(lbl,abs-ser(
                 abs-output(exp),cmds))
       = search-cont(lbl,cmds)
[39] search-cont(lbl,abs-ser(
                 abs-proccall(exp1,exp2),cmds))
       = search-cont(lbl,cmds)
[40] search-cont(lbl,abs-ser(
                 abs-if(exp,cmds1,cmds2),cmds))
       = if(found,<found,cmds3>,
            search-cont(lbl,cat(cmds1,cmds)))
       when <found,cmds3> = search-cont(lbl,cat(cmds2,cmds))
```

The scan of **if**-statements is shown in equation 40. First (in the conditional clause) a continuation is searched in the **else**-branch and the rest of the block. If no continuation has been found the **then**-branch is searched. Remember that the last label counts.

```
[41] search-cont(lbl,
                 abs-ser(abs-while(exp,cmds1),cmds))
       = if(found,<found,cmds3>,
            <found2,cat(cmds2,
                    abs-ser(abs-while(exp,cmds1),
                    cmds))>)
       when <found2,cmds2> = search-cont(lbl,cmds1),
            <found,cmds3>  = search-cont(lbl,cmds)
```

The **while**-construct is treated in equation 41. When a label is encountered in the body of a **while**-loop, the whole loop has to be appended to the remainder of the loopbody after the label. A search is made in the rest of the blockbody for the label. When a continuation is found this is passed on. Otherwise the loopbody is scanned.

```
[42] search-cont(lbl,abs-ser(
                 abs-block(dcls,cmds1),cmds))
       = search-cont(lbl,cmds)
[43] search-cont(lbl,abs-ser(
                 abs-goto(idnt),cmds))
       = search-cont(lbl,cmds)
[44] search-cont(lbl,abs-ser(
                 abs-labldcmnd(idnt,cmd),cmds))
       = if(found,<found,cmds1>,
            <eq(lbl,idnt),abs-ser(cmd,cmds)>)
       when <found,cmds1> = search-cont(lbl,cmds)
```

Equation 44 deals with labeled commands. If the label is found a check is made on the rest of the blockbody to find out if it is the last occurrence of this label. In that case the continuation after the last occurrence is returned. Otherwise the label is compared with the label looked for, and the value of this comparison and the rest of the blockbody are returned.

```
[45] search-cont(lbl,abs-skip) = <false,abs-skip>

end SMALL2
```

Finally, when no corresponding label is found at the end of the block, this result is returned.

3.5 A note on modularity

3.5.1 Auxiliary functions

A problem was encountered with the hiding of auxiliary functions like `cat` in module `SMALL1`. This function is a typical internal construct, needed to make use of an intermediate result in the specification, and in no way an essential feature of SMALL1. When using module `SMALL1` only, one should not be bothered by its existence. However, it is needed in `SMALL2`, so it must be exported or redefined. In this Chapter the problem is ignored by simply exporting everything. The export facility of ASF is too weak to handle such (quite common) situations.

3.5.2 An alternative definition of SMALL1

It is possible to eliminate the auxiliary command `abs-blockend` from the specification of module `SMALL1` through a change in the equations for the evaluation function for series of commands like this:

```
eval(abs-ser(cmd,cmds),senv)
  = eval(cmds,eval(cmd,senv))
```

wherein `eval` also operates on single commands. This has been done in Chapter 2 without problems, since the language there contained no jumps. A specification in this form has a pleasing aesthetic aspect. The evaluation function is able to treat all constructors of commands as primitive operands, not as head or constructor of a list. Thus the specification is both shorter and more symmetric. This specification is given below.

```
module SMALL1'
begin
exports
  begin
    functions
      eval : PROGRAM # ENVELT ->  SENV
      eval : CMND # SENV      ->  SENV -- new
      eval : CMNDS # SENV     ->  SENV
      .
      . as the specification in sect. 3.2 without abs-blockend
      .
      out  :                  -> IDNT
  end
imports SMALL1-Abs-Synt, SMALL1-Tables
variables (identical to the specification in section 3.3.2)
equations
[ 1] eval(program(cmds),input)
       = eval(cmds,table(out,emptylist,
                         table(in,input,null-senv)))
```

```
[2a] eval(abs-assign(exp1,exp2),senv)
       = tablech(idnt,envelt(bval),senv2)
       when <bval,senv1> = evalexpr(exp2,senv),
            <basicval(idnt),senv2> = evalexpr(exp1,senv1)
[3a] eval(abs-output(exp),senv)
       = tablech(out,cat(entry,bval),senv1)
       when <true,entry> = lookup(out,senv1),
            <bval,senv1> = evalexpr(exp,senv)
[4a] eval(abs-proccall(exp1,exp2),senv)
       = eval(abs-block(absdecl-ser(
                          absdecl-const(param,
                            absexp-basicval(bval)),
                          absdecl-skip),cmds1),
              senv1)
       when <basicval(name),senv1> = evalexpr(exp1,senv),
            <true,envelt(absdecl-proc(name,param,cmds1))>
              = lookup(name,senv1),
            <bval,senv2> = evalexpr(exp2,senv1)
[5a] eval(abs-if(exp,cmds1,cmds2),senv)
       = if(bool,eval(cmds1,senv1),
            eval(cmds2,senv1))
       when <basicval(bool),senv1> = evalexpr(exp,senv)
[6a] eval(abs-while(exp,cmds1),senv)
       = if(bool,
            eval(cat(cmds1,abs-ser(
                     abs-while(exp,cmds1),abs-skip)),
                 senv1),
            senv1)
       when <basicval(bool),senv1> = evalexpr(exp,senv)
[7a] eval(abs-block(dcls,cmds1),senv)
       = removeblock(
           eval(cmds1,evaldecl(dcls,blockmark(senv))))
[8a] eval(abs-ser(cmd,cmds),senv)
       = eval(cmds,eval(cmd,senv))
[ 9] eval(abs-skip,senv) = senv
.
. identical to the specification in section 3.3.2
.
[24] cat(abs-skip,cmds) = cmds
end SMALL1'
```

The numbering of the old equations has been retained whenever possible. Unchanged equations retain their number, adapted equations have an "a" appended. Only equation 8a is really new. It replaces the equation describing **abs-blockend** in the first specification.

Other changes fall into two categories. The enclosing `abs-ser` with trailing tail of the program has disappeared everywhere. Secondly, in constructs enclosing an inner series of commands (**if**- and **while**-statements, blocks and with them procedures) the boundaries are delineated by a recursive application of function `eval`. For blocks this results in the superfluity of the marker `abs-blockend`.

3.5.3 A problem with extendint the alternative SMALL1 definition

The modularization of the definition of SMALL2 posed an interesting problem. It was a challenge to make a specification of SMALL1 with functions `eval` and `evaldecl` that could be reused in the specification of SMALL2.

The approach of the preceding section does not allow one to describe non-structured flow of control constructs. An example will serve to illustrate this. Suppose we have a program that executes exactly one jump (to, say, label `lbl`) in some correct evaluation. Then the following equations would hold for this evaluation:

```
eval(abs-ser(abs-goto(lbl),original-tail),senv)
= eval(original-tail,eval(abs-goto(lbl),senv))
= eval(original-tail,
       eval(abs-ser(abs-lbldcmnd(lbl,...),...),senv))
= eval(original-tail,result-senv)
```

In effect the jump-statement is evaluated correctly. However, the obsolete continuation dating back to the state of the evaluation just before the jump is not forgotten. After finishing the whole evaluation in correct order (it has been assumed that it did not contain other jumps), the resulting environment is treated as input for the old tail of the program, clearly an undesired action.

There is no way to avoid this problem when the specification of `SMALL1'` is extended to `SMALL2`. From a model-theoretic point of view modules `SMALL1` and `SMALL1'` have the same initial model (intuitively the language SMALL1). The specification containing equation 8a is stronger than the first specification, hence there are fewer models satisfying it, and the initial model of SMALL2 is not among these.

3.5.4 The alternative definition of SMALL2

Since the evaluation function from the second specification of SMALL1 (module `SMALL1'`) cannot be used to specify SMALL2, the only way to specify an evaluation function for the latter language starting with this specification of SMALL1 is the introduction of another evaluation function. This function will be called `eval2`. A similar solution can be found in [BDMW81].

The adapted specification of SMALL2 is given below. To facilitate comparison the numbering of previous definitions has been retained when possible.

```
module SMALL2'
begin
```

```
exports
  begin
    functions
      eval2            : PROGRAM # ENVELT -> SENV -- new
      eval2            : CMNDS # SENV     -> SENV -- new
      absdecl-lbldcmnd : CMNDS            -> DECL
      .
      . identical to the specification in section 3.4
      .
      progrest         :                  -> IDNT
      abs-blockend     :                  -> CMND -- new
  end
imports SMALL1', SMALL2-Abs-Synt
variables (identical to the specification in section 3.4)
equations
[1b] eval2(program(cmds),input)
       = eval2(cmds,table(out,emptylist,
                          table(in,input,null-senv)))
[2b] eval2(abs-ser(abs-assign(exp1,exp2),cmds),senv)
       = eval2(cmds,senv1)
       when senv1 = eval(abs-assign(exp1,exp2),senv)
[3b] eval2(abs-ser(abs-output(exp),cmds),senv)
       = eval2(cmds,senv1)
       when senv1 = eval(abs-output(exp),senv)
[4b] eval2(abs-ser(abs-proccall(exp1,exp2),cmds),senv)
       = eval2(abs-ser(
                  abs-block(absdecl-ser(
                      absdecl-const(param,
                        absexp-basicval(bval)),
                      absdecl-skip),cmds1),
                  cmds),
                senv1)
       when <basicval(name),senv1> = evalexpr(exp1,senv),
            <true,envelt(absdecl-proc(name,param,cmds1))>
              = lookup(name,senv1),
            <bval,senv2> = evalexpr(exp2,senv1)
[5b] eval2(abs-ser(abs-if(exp,cmds1,cmds2),cmds),senv)
       = if(bool,eval2(cat(cmds1,cmds),senv1),
                 eval2(cat(cmds2,cmds),senv1))
       when <basicval(bool),senv1> = evalexpr(exp,senv)
[6b] eval2(abs-ser(abs-while(exp,cmds1),cmds),senv)
       = if(bool,
            eval2(cat(cmds1,
                      abs-ser(abs-while(exp,cmds1),
                      cmds)),
```

```
                    senv1),
              eval2(cmds,senv1))
        when <basicval(bool),senv1> = evalexpr(exp,senv)
[25a] eval2(abs-ser(abs-labldcmnd(lbl,cmd),cmds),senv)
        = eval2(abs-ser(cmd,cmds),senv)
[26a] eval2(abs-ser(abs-goto(lbl),cmds),senv)
        = eval2(cmds1,senv1)
        when <cmds1,senv1> = jmpcont(lbl,senv)
[27a] eval2(abs-ser(abs-block(dcls,cmds1),cmds),senv)
        = eval2(cat(cmds1,abs-ser(abs-blockend,cmds)),
                deleteprogram(
                  evaldecl(dcls,
                    saveprogram(cmds1,cmds,senv))))
[8b] eval2(abs-ser(abs-blockend,cmds),senv)
          = eval2(cmds,removeblock(senv))
[9b] eval2(abs-skip,senv) = senv
[28] saveprogram(cmds1,cmds,senv)
.
. (identical to the first specification)
.
[45] search-cont(lbl,abs-skip) = <false,abs-skip>
end SMALL2'
```

Unfortunately, function **eval** is rarely reusable. Only commands into which and out of which one cannot jump – this is restricted to assignment-, output- and dummy-statements in SMALL1 – can use the semantics defined with the old function to define the semantics with **eval2**.

All other occurrences of **eval2** have to be defined from scratch, starting with the evaluation of programs, and ending with the reappearance of marker **abs-blockend**. Of course, this specification is similar to the old specification of **eval** in **SMALL1** and **SMALL2**.

The disadvantages of this approach are obvious. The specification of SMALL2 is longer and redoes definitions found in the specification of SMALL1. Also the triviality of the extension has been lost: it is not clear without proof that a SMALL1 program will have the same meaning under the **SMALL1'**- or the **SMALL2'**-specification respectively. The relation between rules 7a and 27a only exists in the sense that they are designed to have the same meaning in specific circumstances, i.e., the evaluation of a SMALL1 program.

However there are also important advantages to this approach. First of all, it is perhaps more realistic: the module **SMALL1'** could come from a library of programming languages as a black box. Also the definition of SMALL1 is more elegant, so the probability of a mistake in this definition is smaller.

3.6 Implementation of the SMALL specification

Our ultimate goal with specifications of this kind is to be able to generate an interpreter or compiler for a programming language, based on an algebraic specification of its semantics. Two attempts have been made to implement the specification presented in this Chapter. Both implementations have been done by hand, but with an open eye for the possibilities to generate them automatically. The scheme specifically designed to be mechanised operates on a class of specifications which is shown to be too restrictive to be of practical use for our purposes. Some comments will be given on the problems concerning automatic translation. A far more thorough treatment of implementation of algebraic specifications can be found in Chapter 5 of [BHK89]. This section aims at providing some intuition concerning this subject. However, the main purpose is to provide some insight into the problems with automatic implementation of interpreters from algebraic specifications.

3.6.1 An "ad hoc" implementation

Term rewriting systems

An algebraic specification can be implemented if it can be turned into a *term rewriting system* ([BK86], [DE84]). This can be done by giving directions to the equations in the sense that A = B is replaced by A $\rightarrow$ B (or A $\leftarrow$ B, but most algebraic specifications have an intuitive direction from left to right). A $\rightarrow$ B has the meaning that term A can be reduced (rewritten) to term B. For B to be a proper reduct of term A it should be closer to a so called normal form (an irreducible term), if A has one. Intuitively, a normal form is the standard, most simple, way to express a certain term.

Similarly, conditional equations of the form

$$A = B \textbf{ when } C_1 = D_1 \text{ ,..., } C_n = D_n$$

are replaced by

$$((C_1 \rightarrow D_1) \wedge \ldots \wedge (C_n \rightarrow D_n)) \rightarrow (A \rightarrow B)$$

niet in ASF implementatie

which means that if C_1 can be rewritten to D_1, C_2 to D_2, etc., then A can be rewritten to B. Of course, as in the standard case A = B above, any of the directions may be wrong.

The theory of term rewriting systems deals with properties like *termination* (every reduction is finite, i.e. after a finite number of steps a normal form is reached) and *confluence* (two diverging finite reduction sequences from the same term have to converge again). In general our algebraic specifications cannot be turned into term rewriting systems with these nice properties. Since we may specify a possibly infinite loop, termination cannot be assured. And treatment of error cases may result in more than one stop criterion. Usually, however, the writer of the specification has a good intuitive working model of his specification in mind, in which these "bugs" are simply ignored.

For our purpose it is good enough if the writer of a specification follows the scheme above in an implementable way. This means that different implementation strategies may yield different implementations in terms of termination behaviour, and in normal forms returned. For the precise criteria avoiding these problems the reader is referred to Chapter 5 of [BHK89]. On the other hand, if the implementation strategy is known and the writer of a specification is prepared to look at the order of his rules with some extra care, a more efficient implementation may result.

A method to represent equations in Prolog

The language Prolog lends itself relatively well to implementing an algebraic specification as a term rewriting system. The arrow in A $\rightarrow$ B can be read as "the analysis of A reduces to the analysis of B". This we can model with a relation **analyse** between terms and their normal forms.

Schematically A $\rightarrow$ B then translates into the Prolog clause:

```
analyse(A, Res) :- analyse(B, Res).
```

which reads: "the analysis of term A has result Res when the analysis of term B has result Res". This crude scheme will need modification, however, to deal with evaluation of arguments of term A that have to be known first. With the same provision for both term A and terms C_i, rules of the format $((C_1 \rightarrow D_1) \wedge ... \wedge (C_n \rightarrow C_n)) \rightarrow (A \rightarrow B)$ translate to:

```
analyse(A, Res)
    :- analyse(C1 , Res1),
       ...,
       analyse(Cn , Resn),
       analyse(B, Res).
```

Since conditions $C_i \rightarrow D_i$ may interact in the sense that one defines an intermediate result for another, during translation their order may have to be changed to provide for the correct interdependency.

Sometimes no constructive translation of the **when**-part of the specification exists. This happens when the clause is used to simulate an existential quantifier. Hence the wish to produce implementable specifications automatically will pose constraints on the class of allowable specifications.

The "ad hoc" implementation

The specification of SMALL2 from sections 3.3 and 3.4 has been implemented along the lines indicated above. This posed only minor difficulties, though it indicated some possible problem areas.

The main trouble spot from the point of view of implementation is the **when**-clause. Existential quantifiers had to be eliminated. Most of them were just aliases for longer expressions. The specification contained some trivial cases of true quantification, for instance variable `bool` in equation 5 (section 3.3) is quantified over

the sort BOOL, the equation has no meaning when evaluation of expression **exp** would yield something else. A close operational translation has been made in these cases, which posed no difficulties. To recognise aliases more easily introduction of a keyword reserved for abbreviation might allow enough flexibility while avoiding confusion. True existential quantification could then be ruled out without problem.

More thought went into the correct order of the evaluation of the conditions. It looks feasible to let the order of specification be the order of evaluation. This follows closely the intuitively attractive bottom-up approach in writing conditions. Alternatively, the reverse order of specification, corresponding to the top-down approach for writing, could be chosen.

The evaluation scheme presented so far cannot handle terms with terms as arguments. These terms fall into two categories per argument. The easiest and more frequent form is

```
f(X) = g(h(X))
```

which is equivalent to

```
f(X) = g(Y)
   when h(X) = Y
```

and can be evaluated as such. Sometimes, however, no intermediate result can be found, as in

```
if(Test,Then-part,Else-part) = Res
```

where for instance the **Then-part** might be ill-defined if the **Test** evaluates to **false**. These cases have been solved by splitting such equations as follows:

```
if(Test,Then-part,Else-part) = Then-part
   when Test = true
if(Test,Then-part,Else-part) = Else-part
   when Test = false
```

A more general discussion of techniques for delaying the evaluation of certain arguments follows in the sections below.

3.6.2 The automatic scheme of Drosten and Ehrich

An automatic translation method for algebraic specifications has been proposed by Drosten and Ehrich [DE84]. Their scheme is an innermost reduction scheme: first all arguments of a function are brought into normal form before the function as a whole is tackled. The check whether arguments are in normal form has two distinct disadvantages for implementation purposes.

- Normal forms may be checked over and over again, which decreases efficiency. This may be solved by creating a *cache* of known normal forms. However, in general such a *cache* will rapidly become very large. A much better solution is the addition of a normal form flag to every term, so they can be recognized "at a glance".

- More serious is the problem that this scheme is not optimal with respect to the termination behaviour of the resulting rewrite system. For instance, this scheme cannot cope with non-strict functions (functions which can successfully be evaluated even though one or more of their arguments are still unknown). The prime example in this category is the function:

  `if(Test, Then-part, Else-part)`

  in which the `Else-part` can be disregarded if `Test` evaluates to `true`, and the `Then-part` if it is `false`. A function like this is needed to specify evaluation of loops in a language. The loop-test then corresponds to the test in the `if`-function above, the evaluation of the loop-body *followed by the loop again* to, e.g., the then-part, and termination to the else-part. An innermost reduction of this function would result in the reduction of the then-part. This reduction in turn will (after reduction of the loop-body) result in the reduction of the full clause above, containing the same then-part again, thus producing a potentially infinite derivation. Of course the then-part should only be rewritten when the test evaluates to `true`.

For our purposes improvement is needed in the scheme of Drosten and Ehrich on efficiency (the number of reductions performed and the stack space used) and termination. The next section introduces an alternative reduction strategy principle.

3.6.3 Outermost reduction strategies

The converse of innermost reduction strategies are outermost reduction strategies. These reduction strategies postpone evaluation and try to do as little work as possible. Hence they are alternatively described as lazy evaluation.

When applying an outermost reduction strategy one first tries to reduce a term as a whole. If this does not succeed, an attempt is made to perform at least one (outermost) reduction step of one of its arguments. This method is repeated until no further reductions can be performed.

Outermost reduction strategies differ in the number and order of arguments that are reduced when the outermost function cannot be reduced as it stands. Leftmost-outermost reduction, e.g., only reduces the leftmost argument that can be reduced. This reduction strategy would be the most efficient strategy for the evaluation of the `if`-function in the preceding section. However, should the function be changed to `if(Then-part, Else-part, Test)` evaluation may never end again. (Viz. the `Then-part` contains an infinite loop when the `Test` reduces to `false`.) Since we cannot assume prior information about the order in which arguments should be evaluated another strategy is needed.

An outermost reduction strategy with reasonably broad application fields is parallel outermost reduction, i.e., if a term cannot be reduced, all arguments are reduced in parallel. This reduction scheme has a better termination behaviour than the scheme of Drosten and Ehrich. It will perform at most the same number of one-step reductions as the innermost scheme.

In certain cases the normal form of a proper subterm has to be found before a rule can be applied. If this reduction takes several steps the reduction of the term as a whole acts like a yo-yo: no rule can be applied to the term so the internal arguments are examined and one step is applied. The scheme calls for another test at top level (which fails) and the whole term has to be dissected again. Here the innermost scheme would be more efficient.

Optimal termination behaviour can be reached when the user indicates which arguments are essential ("needed redex") for reduction to progress and which arguments can or must be delayed. Such an attitude transfers responsibility to the user for the choice of reduction strategies.

3.7 Conclusions

The prime question to be answered in this Chapter is whether an elegant algebraic specification can be given of the most unstructured of the classical program features: the jump. The present specification is somewhat longer than the specification in *denotational semantics* by Gordon [Gor79]. It is felt, however, that the algebraic specification is at least as legible as the denotational specification. In addition to this, our specification addresses the problem of building a language definition in layers, thus reusing and extending language constructs defined in previous layers.

Progress is being made in the field of modularity of specifications. Work by Bergstra, Heering and Klint on *module algebra* [BHK90] provides formal tools to reason about import/export relationships. The problems encountered are largely circumvented in the present Chapter, with the notable exception of the more elegant definition of function `eval` in section 3.5.

The question of efficient implementation of algebraic specifications is still an open problem. The solutions suggested in section 3.6 are either to restrict the class of allowable specifications or to give the writer more responsibility for the termination behaviour of the term rewriting system derived from his specification. This subject is treated in Chapter 5 of [BHK89] in more detail, but an optimal trade-off cannot be given yet.

Chapter 4

Implementation of Modular Algebraic Specifications

The foundation of implementation of algebraic specifications in a modular way is investigated. Given an algebraic specification with visible and hidden signature an *observing* signature is defined. This is a part of the visible signature which is used to observe the behaviour of the implementation.

Two correctness criteria are given for the implementation with respect to the observing signature. An algebraic correctness criterion guarantees initial algebraic semantics for the specification as seen through the observing signature, while allowing freedom for other parts of the signature, to the extent that even final semantics may be used there. A functional correctness criterion allows one to prove the correctness of the implementation for one observing function in Hoare Logic. The union over all observing functions of such implementations provides an actual implementation in any programming language with semantics as described above.

4.1 Introduction

An algebraic specification is a mathematical structure consisting of sorts, functions (and constants) over these sorts, and equations describing the relation between the functions and constants. It is a convenient tool to specify static and dynamic semantics of programming languages, see e.g. Goguen and Meseguer [GM82], [GM84], [MG85]) for more detail on algebraic specification, and [BHK89], [BDMW81] for examples. The implementation of an algebraic specification usually consists of the conversion of the equations into a *term rewriting system*, either directly or through the completion procedure of Knuth-Bendix. More details can be found in [HO80] and [ODo85]. The performance of such an implementation is rather slow in general, compared with algorithms written in conventional programming languages, while the specification must have certain properties in order to be implemented in this way at all. The aim of this Chapter is to provide another implementation strategy, based on pre- and postconditions, allowing the application of more classical programming and optimization techniques.

4.1.1 Modular algebraic specifications

Algebraic specifications have been introduced to provide a description style for data types in a mathematically nice way. The mathematical notion of a (many-sorted) *algebra* used here is a structure consisting of carrier sets and typed functions (including constants) over these sets, together with a set of equations, specifying the behaviour of the functions. The combination of a set of sorts (the names of the carrier sets) and a set of functions (which include constants, unless stated otherwise), is called the *signature* of the algebra.

Prompted by both theoretical and practical considerations, the algebraic specifications studied in this Chapter have additional organization primitives. Central issue is the *modular structure* imposed on the algebraic specifications. An algebraic specification can import another algebraic specification as a *module*, meaning that it adds the sorts, functions and equations of the imported specification to its own. Sorts or functions with the same name are only allowed when they are the same (i.e., they originate as the same sort or function in the same module), otherwise they must be renamed.

The modular approach naturally leads to two other primitives, a *parameter* mechanism, and the occurrence of *hidden* (local, auxiliary) sorts and functions. Hidden sorts and functions are used in the equations of the module in which they are defined, but they are not included in the exported or *visible* sorts and functions. Only the latter are included in the algebra associated with the module. Hidden sorts and functions make it easier to write many specifications by providing local definitions. Also, they are necessary to specify properties needing an infinite number of equations (when defined without hidden sorts and functions) in a finite way (see Bergstra and Tucker [BT83], [BT82]).

The equations used are *conditional equations*, i.e. equations which are valid only when certain conditions are satisfied. The semantics provided is *initial algebra*

semantics. However, for reasons of efficiency implementations can have modified semantics in this Chapter. Initial algebra semantics are described by the catch-phrases 'no junk' (all elements of the specified sorts can be reached via the specified functions) and 'no confusion' (everything which is equal in the algebra can be proved equal with the equations provided). These semantics are usually intuitively clear.

4.1.2 Implementation of algebraic specifications

Once an algebraic specification has been written there is no clearcut way to derive a working program from it. In general, any model of the specification can be seen as an implementation. Some of the more usual choices are presented below.

A strategy followed quite often to implement a model satisfying initial semantics (an initial model) is to transform the specification into a *term rewriting system*. The easiest way to do this is to give every equation a direction, say from left to right, and to view the set of directed equations as a set of *rewrite rules*, transforming one term over the signature into another. This procedure can be found in various places in the literature ([BK86], [BHK89], [DE84], [FGJM85], [GMP83], [ODo85], [Wal91]), but the success of this method depends on the properties of the directed version of the (in principle undirected) set of equations, combined with the technique used for rewriting. Turning the 'direction' of an equation around (writing B=A instead of A=B) or writing the equations in a different order may have significant consequences for the behaviour (both in speed and in termination) of the implementation, while the specification has not been changed, except textually.

An additional problem is the question what to do with the modular structure when fitting a modular specification in a term rewriting system. Transparent semantics can be obtained by a *normalization step* (as described by Bergstra, Heering and Klint [BHK89], [BHK90]), flattening all imports into one module (renaming hidden functions and sorts where necessary). The term rewriting approach above can be applied to the normalized module. It may be debated whether the loss of the structure in the specification is sufficiently motivated by the transparency of the semantics.

The present Chapter aims at a more module-oriented implementation, giving semantics to implement an *observing signature* (a signature through which one can observe the visible signature, of which it is a subsignature) in a functional way, using descriptions of the observing functions in *Hoare Logic* (see e.g. the text books [LS84], [Bac86]). The significance for the semantics of the import construct will be touched upon briefly. The main advantage of this approach is that it permits the implementation of modules in a, possibly low-level, efficient way from the high-level specification. This allows the construction of a library of efficient basic modules upon which more sophisticated algebraic specifications may depend.

4.1.3 Related work

An overview of the state-of-the-art in implementation relations can be found in section 1.2.3 of the survey book [COMPASS91]. The implementation notion presented

here is in the category of simulation, i.e. a specification SP1 implements specification SP2 if the behaviour of SP2 simulates the behaviour of SP1. The choice for our approach is motivated as follows.

An important approach towards implementation is in terms of models. Implementation techniques for pure initial semantics are burdened with the obligation to implement the initial algebra semantics faithfully. This generally slows down the implementation, since often an initial specification demands too much detail, as has been discussed by Baker-Finch [Bak84]. Realization functions as proposed by Ehrig and Mahr [EM90] solve some of those problems by describing the implementation in 'specification morphisms', i.e. functions mapping one specification to another. This approach is still felt to be too restrictive in specific cases.

On the other side of the spectrum the category theory approach towards specifications provides elegant general solutions, see, e.g., [ST88] and [Sch87]. However, this generalization abstracts too much from the implementation details needed for an efficient implementation.

Meseguer and Goguen [MG85] also provide an implementation criterion for algebraic specifications. They focus on observable sorts, while the present Chapter works with observing functions. Their approach is a special case of the approach presented here. They retain initial algebra semantics for their specifications but loosen the restriction on the models for implementation. In the present Chapter the semantics are modified.

The recent work by Bernot, Bidoit and Knapik [BBK92] expands on the sort observation notion and is even more general than the implementation notion presented here. [MG85] uses all terms in sets of sorts to observe, our observations use sets of terms generated by a set of functions, and [BBK92] uses general sets of terms. Other recent work by Hennicker [Hen91] is rather more cumbersome since it requires the specification of a term algebra of observing terms. Those terms again do not necessarily correspond to all terms of a function, so those two approaches allow for the implementation of possibly dangerously unexpected partial functions.

There is a strong resemblance to *abstract data type* theory as practiced in the verification of correctness of programs (cf. Jones [Jon80]). After all, an algebraic specification is a nice way to describe a data type. While the specifications look similar, the point of view is different. Constructor functions (i.e. functions describing the data type) really construct the type in algebraic specifications, while they only serve as a description tool, a convenient way to generate all elements of the type, in [Jon80].

Techniques which use *term rewriting systems* have the advantage of allowing (semi-)automatic translation schemes, but pay the price with severe restrictions on the set of equations allowed. Perhaps the overhead of a completion procedure for generation of rewrite rules is needed, e.g. the Knuth-Bendix procedure (see [HO80] and [ODo85] for more detail). The technique presented here allows for faster implementations, but does not support automatic translation.

4.1.4 An outline of this Chapter

In the next section brief introductions to both specification formalisms used (the algebraic specification formalism ASF and Hoare Logic) are given.

Section 4.3 starts with an example to illustrate certain disadvantages of the initial algebra approach to motivate the theoretical framework leading to an algebraic implementation notion in the second half. An example giving an implementation according to this notion follows in section 4.4, which may be read before section 4.3.2 to get the flavour, or in the order provided to convince oneself of the rigour of the approach.

The functional implementation notion is described as an extension of the algebraic notion in section 4.5, preceded by an example to show the insufficient strength of the latter notion for our purpose. The example of section 4.4 is implemented in an imperative language in the following section according to this notion. Some final remarks are made in section 4.7.

4.2 The formalisms informally

4.2.1 Horizontal composition in algebraic specifications

The algebraic specifications in this Chapter are presented in the specification formalism ASF introduced in Chapters 1 and 3.

The semantics of a module is defined by the initial algebra over the export (visible) signature and the function `if`, as built-in in ASF. Imported modules are 'normalized', i.e. all hidden functions are made unique by renaming if necessary, and all equations are taken over with the same renaming.

Hence *horizontal composition*, the combination of two or more imported specifications, is semantically rather easy. But this does not allow for separate implementation: all is lumped together. The implementation strategy in the sequel does allow for proper separate implementation, at a cost of complexity in semantics. This movement away from initial semantics is argued to be necessary for efficient implementation in section 4.3.1.

4.2.2 Hoare logic and abstract data types

Hoare logic is a well-known technique to describe the behaviour of programs in both imperative and functional languages. It has found its way into various text books, e.g. [LS84] and [Bac86], which provide more rigour. Briefly, Hoare logic allows one to write {P} S {Q}, meaning that evaluation of program S in a state in which *precondition* P holds results in a state in which *postcondition* Q holds. These conditions describe the state vector, i.e., the variables and their contents, of the program. Various proof rules and proof techniques are available to verify such a program.

In this Chapter some functions specified in an algebraic way will be specified in an equivalent Hoare logic way by giving conditions on its input and output. Such a specification is independent of the actual implementation program, which may be changed (and preferably optimized). Since Hoare logic techniques can be formulated for many languages the ultimate program could be written in any appropriate language, at some cost in interfacing. Hence a large degree of language independence for the implementation is achieved, allowing various kinds of optimization strategies.

One way to interpret an algebraic specification is as a high level specification of an *abstract data type*. Hence the implementation strategy for algebraic specifications presented here bears a more than casual resemblance to the theory of implementation of abstract data types. An abstract data type is some type together with a set of functions on the type. An implementation is a more concrete (i.e., closer to machine level) type with a corresponding set of functions which model the abstract type and functions. This is done by providing a translation back and forth between the abstract and concrete types, such that the abstract functions are simulated correctly by the combination of the translation to the concrete type, application of the concrete function and the translation back to the abstract type (cf. Jones [Jon80]).

The scheme in the Chapter basically relaxes the translation conditions for all terms in the initial model of the algebraic specification by demanding translations for specific terms only. This stems from the functional orientation: only the input terms need to be translated and only the output terms need to be translated back. Section 4.5 provides more detail.

4.3 Algebraic implementation

4.3.1 Initial algebra semantics and reusability

The question we want to consider is the following. Suppose we have an algebraic specification and we want to make in some way an efficient implementation for further use by someone else, as in a library. What is the interaction between efficiency and semantics?

Initial algebra semantics have much in favour. They are characterized by 'no junk', i.e., it is clear which objects exist, and 'no confusion', i.e., closed terms (terms without variables) are only equal if they can be proved equal using equational logic. While these two characterizations are clearly desirable in many circumstances they cannot always both be met.

The 'no confusion' condition generates overspecification in the sense that terms might be distinguished from each other without necessity. If the writer of a specification does not care about whether two terms are equal or not (in the common case that their usage is identical), and hence does not specify their equality, they are unequal. This puts a burden on the implementor of the specification to provide this inequality, disallowing a possibly more efficient identification. Since it is not possible to specify which terms must be unequal the only tool available is the precise definition of the opposite property – equality – by extending the number of equations. This puts a burden on the shoulders of the specifier, who has to provide these additional equations. While the extra amount of work is undesirable it is also not clear in general what additional equations are necessary and whether a sufficient set can be found at all. For discussions see [Bak84], [Kam83] and [Wan79].

An example will serve to illustrate this. A very common datatype is the bounded array. Suppose a specifier wants to define an array of natural numbers of length 10, indexed from 0 to 9. It should be possible to put natural numbers into the array at certain indices and to retrieve them again, getting the latest entry for that index. Initially, all entries are set to 0, and of course they can be reset to this value, simply by entering a 0 in every slot. Out of bound indices are simply ignored. In practice one would probably want to have a more robust version, but this will be at the cost of a longer specification. The following specification is a natural way to describe such a bounded array.

```
module BoundedArray
begin

exports
        begin
        sorts ARR
        functions
                newarr  :                   -> ARR
                put     : NAT # NAT # ARR -> ARR
                maxindex:                   -> NAT
                get     : NAT # ARR         -> NAT
```

```
        end

imports Booleans, Natnumbers

variables
        i,j,v : -> NAT
        arr   : -> ARR

equations
[1] maxindex = 9
[2] get(i,newarr) = 0
[3] get(i,put(j,v,arr)) = if(greater(i,maxindex),
                             0,
                             if(equal(i,j),
                                v,
                                get(i,arr)))
[4] put(i,v,arr) = arr
                   when greater(i,maxindex) = true
end BoundedArray
```

This specification contains just about what one wants to specify if the output behaviour of function **get** is the only thing of concern. Equation 1 fixes **maxindex** and equation 4 says that additions above this key have no effect. Equations 2 and 3 describe the behaviour of function **get**. If a user imports this module and restricts the use of the result of functions in the module to **get** only, it will behave as a bounded array, so no need is felt to extend the specification.

The problem is, that function **put** is hardly specified. This is natural, since the writer of module **BoundedArray** concentrated on **get**, the function he wanted to specify. Indeed, in terms of functionality of **get** there is nothing wrong with the specification of **put** as it is. However, since we are using initial semantics, it is possible for someone importing module **BoundedArray** to extend the number of functions on sort **ARR** with

```
sum      : NAT # ARR        -> NAT
```

and to add the following equations:

```
[5] sum(i,newarr) = 0
[6] sum(i,put(j,v,arr)) = if(greater(i,maxindex),
                             0,
                             if(equal(i,j),
                                add(v,sum(i,arr)),
                                sum(i,arr)))
```

This new function makes a summation over *all* entries ever put into a certain index value of the array. It is a well-defined new function in the sense that no unexpected identifications of terms in other sorts than **ARR** (which we are redefining)

occur. Of course, the writer of module **BoundedArray** intended to have only the last entry at hand.

How can the specification be remedied? The straightforward way to get rid of function **sum** is to put restrictions on the terms of sort **ARR** allowed. Old entries should be forgotten. The following equation specifies that:

```
[7] put(i,v1,put(j,v2,arr)) = if(equal(i,j),
                                  put(i,v1,arr),
                                  put(j,v2,put(i,v1,arr)))
```

Now the definition of **sum** would result in undesired identifications in sort **NAT**. With only the equations 5 and 6, obviously **sum(put(1, 2, put(1, 3, newarr)))** = 3. With equation 7, however, also **sum(put(1, 2, put(1, 3, newarr)))** = **sum(put(1, 3, put(1, 2, newarr)))** = 2, so 2 = 3. Hence the function **sum** defined in this way is ruled out (it is still allowed to define it like this, but the resulting module will have unintended properties).

Addition of this one equation seems to be fine, but how can we be sure that the story ends here? This requires a non-trivial proof, for instance giving an isomorphism between the initial algebraic model generated by the specification, and the data type to be modeled.

In the example, the combination of equations 1 through 4 and 7 does not allow that. The terms **newarr** and **put(7,0,newarr)** cannot be proved equal by these equations, though they both describe the array containing exclusively zeros. Of course, the equation

```
[8] newarr = put(i,0,newarr)
```

can be added , but then the question whether the set of equations (now 1 through 4 and 7 and 8) fixes what was intended returns again.

Actually, these six equations are sufficient. A proof could proceed as follows. First a set of canonical forms is defined, e.g. the set of terms with at most one **put** for every key in order of the keys and without entries of value **0**. Obviously a bijection between the set of canonical forms and arrays of length 10 can be found. It also can be proved that every well-formed term of sort **ARR** is equationally equal to exactly one of these canonical terms. So the term model has exactly the same structure.

The surest way to supply an answer in a structure defined by initial algebra semantics is to add a constructor function actually making an isomorphic image of the object wanted. In the example this is a bounded array of length **maxindex**+1, so for instance a new constructor function with exactly 10 'holes' of type **NAT** would do it. Then additional specifications must be provided in terms of this constructor function. In the example this would become a function:

```
arr: NAT # NAT # NAT # NAT # NAT #
     NAT # NAT # NAT # NAT # NAT -> ARR
```

and in addition to equations 1 through 4 the equations (**v0**...**v9** are variables of sort **NAT**):

```
[n0] newarr = arr(0,0,0,0,0,0,0,0,0,0)
[p0] put(0,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v,v1,v2,v3,v4,v5,v6,v7,v8,v9)
[p1] put(1,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v0,v,v2,v3,v4,v5,v6,v7,v8,v9)
            ...
[p9] put(9,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v)
```

Apart from the question whether such an *ad hoc* solution can be found in general this is contrary to the amount of detail one wants to specify algebraically. For this two important considerations can be given, one philosophical and one practical:

- Algebraic specification is a higher level programming formalism. While the formalism is powerful enough to express a computer up to bit level if necessary, this is a waste of effort. There are more than enough lower level programming languages already.

- An algebraic specification (and indeed any specification) is made with a certain use of the objects to be specified in mind. This use is what has to be specified in detail, since that is what has to be implemented. Other details specified are peripheral in the sense that one might have chosen another description. The fewer details fixed in these peripheral specifications the more freedom left to an implementor for optimizing it. The choice of models for implementation should not be restricted to one model (up to isomorphism), but rather be as broad as the class of all models — as far as they can be implemented. So demanding the implementation of initial algebra semantics is too restrictive.

This practical point is illustrated in the example. Module **BoundedArray** is written with external use of only the function **get** (and perhaps the constants **newarr** and **maxindex**) in mind. However, equation 6 describes a new function in terms of function **put**. So **put** becomes a genuine constructor function, while we only wanted a convenient function to enter natural numbers in a "behind-the-screen" data representation.

The exact form of the data representation is of little interest to the user of function **get**, as long as this use is not affected. In this example probably a simple array of length 10 is what you want. But changing the value of **maxindex** to some large number might make a sparse array approach or a hash-table the better implementation.

For the remainder of the Chapter we distinguish three important subsets in the signature of an algebraic specification:

- The **visible** signature which generates the terms existing in the specification for the outside world.

- The **hidden** signature, which is necessary to obtain finite initial algebra specifications on the one hand and handy as a shorthand mechanism and alternative

data description on the other hand. The complete signature is the union of the visible and the hidden signature.

- The **observing** signature, which restricts the terms generated by the visible signature. It contains the functions through which visible terms may be used and the sorts with terms which may be used as observing terms. A term is an observing term if both the head function and its sort are in the observing signature. This signature is the subset of the visible signature one wants to be implemented as specified. In the example this is the signature with sort `NAT` and function `get`.

The choice of the functions and sorts in the observing signature depends on the goal one has in mind for the specification. Enlarging this signature enhances the possibilities for use but restricts the freedom of the implementor. So one can opt for a fast, but narrowly applicable implementation, or for a more generally usable, but slower implementation.

Of course, the speed of a certain function in an observing signature is not only dependent on the signature but also on the implementations of other (not necessarily observing) functions. One can for instance trade the speed of an insertion function for the speed of a retrieval function by gearing the underlying data structure (at this level of abstraction represented by the visible functions that are not observing and the hidden functions) to the preferred task.

The consequences of this tripartition for the theory are investigated in the next section.

4.3.2 A theory of algebraic implementation

This section is devoted to the development of a theory for the subsequently introduced notion of *algebraic implementation* with respect to an *observing signature*. Roughly speaking two algebraic specifications are algebraic implementations of each other when the behaviour of the observing functions is the same in both specifications. An annotated example is provided in section 4.4. The reader may wish to read the example first, referring back to notations and details in this section when necessary.

4.3.2.1 Notations (algebraic specifications)

In the rest of the Chapter the following conventions are used:

a. A **signature** Σ is a tuple (S,F) in which S is a set of sorts and F a set of typed functions. (Note that there is no intrinsic relation between the sorts in S and F.) Often an element of F is denoted by its name only, providing typing when necessary. Two functions with the same name, but different typing are different functions.

b. A **complete signature** Σ = (S,F) is a signature in which for all $f : s_1 \times ... \times s_k \rightarrow s \in$ F holds that all sorts in the typing of f are available in S, so $s_1, ..., s_k, s \in$ S.

c. For a signature Σ, $T(\Sigma)$ is the **set of closed terms**; $T_s(\Sigma)$ is the subset of terms of sort s from $T(\Sigma)$.

d. Union, intersection, and inclusion are defined for signatures Σ_1, Σ_2 ($\Sigma_i = (S_i,F_i)$), as:

$$\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2)$$
$$\Sigma_1 \cap \Sigma_2 = (S_1 \cap S_2, F_1 \cap F_2)$$
$$\Sigma_1 \subset \Sigma_2 = S_1 \subset S_2 \wedge F_1 \subset F_2$$

e. An **algebraic specification** is a tuple (Σ_V, Σ_H, E) with

- $\Sigma_V = (S_V, F_V)$ a complete signature (the **visible** signature),
- $\Sigma_H = (S_H, F_H)$ a signature (the **hidden** signature) such that $\Sigma_V \cup \Sigma_H$ (the **internal** signature) is a complete signature, and
- E a set of equations over $T(\Sigma_V \cup \Sigma_H)$, enriched with typed variables of sorts from $S_V \cup S_H$.

f. Let (Σ_V, Σ_H, E) be an algebraic specification and let $t, t' \in T(\Sigma_V \cup \Sigma_H)$. For an equation $e \in E$, t **and** t' **are equal through direct substitution in equation** e (i.e., in one step) is written as:

$$t =_e t'.$$

g. t **and** t' **are equationally equal**, i.e., equal through zero or more direct substitutions in one or more equations from E, is written as:

$$t =_E t'.$$

transitiviteit

4.3.2.2 Definitions (Σ_O-observability and -equality)

Let (Σ_V, Σ_H, E) be an algebraic specification and $\Sigma_O = (S_O, F_O)$ (the **observing** signature) a signature such that $\Sigma_O \subset \Sigma_V$.

a. The **set of closed Σ_O-terms over Σ_V**, also called the **set of observing terms** is the set of terms in $T(\Sigma_V)$ of sort in S_O and head function symbol in F_O. It is defined as:

$$T(\Sigma_O, \Sigma_V) = \{t \in T(\Sigma_V) \mid \exists f \in F_O, s \in S_O, f : s_1 \times ... \times s_k \rightarrow s, \exists u_1 \in T(\Sigma_V) ... \exists u_k \in T(\Sigma_V) \; t = f(u_1, ..., u_k)\}.$$

(The =-sign in the formula stands for syntactic equality.) The **set of closed Σ_O-terms over Σ_V of sort** s is written as $T_s(\Sigma_O, \Sigma_V)$.

Note that it is possible to have functions in F_O whose output sort is not in in S_O, or sorts in S_O which cannot be reached from F_O. This choice of notation is motivated by the function-oriented approach of this Chapter. By choosing

a set of observing functions F_O and all visible sorts S_V for S_O, all sorts in S_V which cannot be reached do not influence $T(\Sigma_O, \Sigma_V)$. For reasons of symmetry the definition is formulated in such a way that one can also restrict the sorts and not the functions, as will be done in point e below. Alternatively, it is possible to define S_O as the set of sorts in the range of F_O. This does not affect the theory.

b. Where no confusion can arise the following abbreviations are used:

$$\begin{aligned} T_O &= T(\Sigma_O, \Sigma_V) \\ T_V &= T(\Sigma_V) \\ T_{s,O} &= T_s(\Sigma_O, \Sigma_V) \\ T_{s,V} &= T_s(\Sigma_V) \end{aligned}$$

c. A **context** (for sort s) $T(\bullet_s)$ is a term with a missing subterm of sort s. The **empty context** (i.e., a context in which the top term is missing) is written as $\bullet_s$.

A term $t \in T_{s,V}$ is Σ_O**-observable** if and only if there exists a context $T(\bullet_s)$ such that $T(t) \in T_O$;

a Σ_O-observable term $t \in T_{s,V}$ is **directly** Σ_O**-observable** if and only if $t \in T_O$ (hence the empty context $T(\bullet_s) = \bullet_s$ satisfies $T(t) \in T_O$);

a Σ_O-observable term $t \in T_{s,V}$ is **indirectly** Σ_O**-observable** if and only if $t \notin T_O$ (hence the empty context T does not satisfy $T(t) \in T_O$).

d. Σ_O**-equality** (i.e., equality with respect to observations through the observing signature Σ_O) is defined as follows for two terms $t, t' \in T_{s,V}$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow \forall T(\bullet_s)[T(t), T(t') \in T_O \rightarrow T(t) =_E T(t')].$$

Where no confusion can arise $\sim_{E,\Sigma_O}$ is abbreviated to $\sim_O$.

e.
- Let $f \in F_V$. A term $t \in T(\Sigma_V)$ is f**-observable** if and only if t is $(S_V, \{f\})$-observable; two terms in $T(\Sigma_V)$ are f**-equal** if and only if they are $(S_V, \{f\})$-equal.
- Let $s \in S_V$. A term $t \in T(\Sigma_V)$ is s**-observable** if and only if t is $(\{s\}, F_V)$-observable; two terms in $T(\Sigma_V)$ are s**-equal** if and only if they are $(\{s\}, F_V)$-equal.

Since the definition in case c ignores unreachable sorts and functions to unavailable sorts, only terms with head symbol f in the first case, and with range s in the last, are relevant.

The notion of observability via a sort corresponds to the notion in [MG85], and is underlying the behavioural equivalence notion in [ST87]. In the latter paper, the observational equivalence notion is very general, since it is parameterized with the logic used to reason about observations. Thus Σ_O-equality corresponds to observational equivalence under conditional equational logic in the terms of [ST87]. By concentrating on one logic more can be said about the implementation in the present Chapter.

4.3.2.3 Some facts about Σ_O-observability and -equality

In final algebra semantics terms are equal unless they can be proved different, so in models with final semantics there is the maximum amount of 'confusion' consistent with the inequalities which must exist in the model. As such, Σ_O-equality is a notion from final algebra semantics. If you want to show that two closed terms are different you have to find a context for which these terms behave differently, thus proving their inequality. If no such context can be found the terms cannot be distinguished from each other. In initial semantics, on the contrary, they are distinguished unless they are equationally equal, in other words, unless they can be transformed into each other via equations from E, thus proving their equality.

Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ be an algebraic specification, $\Sigma_O \subset \Sigma_V$, and $t, t' \in T_{s,V}$, then the following facts hold:

a. $T(\Sigma_V, \Sigma_V) = T(\Sigma_V)$

   Observing through the visible signature gives all visible terms. This follows immediately from definition 4.3.2.2.a.

b. If $\Sigma'_O \subset \Sigma_O$ then $T(\Sigma'_O, \Sigma_V) \subset T(\Sigma_O, \Sigma_V)$.

   A smaller observing signature results in a smaller set of observing terms. Again this follows from definition 4.3.2.2.a.

c. If $\Sigma'_O \subset \Sigma_O$ then $t \sim_{\mathrm{E},\Sigma_O} t' \rightarrow t \sim_{\mathrm{E},\Sigma'_O} t'$.

   This follows from the definition of Σ_O-equality and fact b, since there are fewer contexts in $T(\Sigma'_O, \Sigma_V)$ than in $T(\Sigma_O, \Sigma_V)$ to show the difference between t and t'.

d. If t and t' are not Σ_O-observable they are Σ_O-equal.

   Since there is no context to show the difference between t and t' this follows from the definition.

e. If t is Σ_O-observable and t' is not then they are Σ_O-equal.

   The argument for fact d holds here also.

f. It should be noted that in cases d and e both $t \sim_{\mathrm{E},\Sigma_V} t'$ and $\neg t \sim_{\mathrm{E},\Sigma_V} t'$ can be true. Take for example:

$$\begin{aligned} \Sigma_V &= (\{s\}, \{a, b\}) \text{ with } a, b \in s, \text{ and} \\ \Sigma_O &= (\{s\}, \{a\}) \text{ for case e, or} \\ \Sigma_O &= (\{s\}, \emptyset) \text{ for case d.} \end{aligned}$$

   When the set of equations is empty the following holds:

$$a \sim_{\emptyset,\Sigma_O} b \text{ and } \neg a \sim_{\emptyset,\Sigma_V} b,$$

   while $\mathrm{E} = \{a = b\}$ results in:

$$a \sim_{\{a=b\},\Sigma_O} b \text{ and } a \sim_{\{a=b\},\Sigma_V} b.$$

g. It has been mentioned already in definition 4.3.2.2.e, that the definition of observability ignores unreachable sorts and functions to unavailable sorts. More formally (*ran* stands for range):

- Let $F_O \subset F_V$. Let $ran(F_O) \subset S \subset S_V$.
  A term $t \in T(\Sigma_V)$ is $(ran(F_O), F_O)$-observable if and only if t is (S, F_O)-observable if and only if t is (S_V, F_O)-observable; two terms in $T(\Sigma_V)$ are $(ran(F_O), F_O)$-equal if and only if they are (S, F_O)-equal if and only if they are (S_V, F_O)-equal.
- Let $S_O \subset S_V$. Let $\{f \in F_V \mid ran(f) \in S_O\} \subset F \subset F_V$.
  A term $t \in T(\Sigma_V)$ is $(S_O, \{f \in F_V \mid ran(f) \in S_O\})$-observable if and only if t is (S_O, F)-observable if and only if t is (S_O, F_V)-observable; two terms in $T(\Sigma_V)$ are $(S_O, \{f \in F_V \mid ran(f) \in S_O\})$-equal if and only if they are (S_O, F)-equal if and only if they are (S_O, F_V)-equal.

The following lemma states that the initial algebraic structure is retained on directly observable terms. So only indirectly observable and unobservable terms can lose their initial behaviour. It follows immediately (corollary 4.3.2.5) that no restriction on the observability (i.e., $\Sigma_O = \Sigma_V$) retains the initial algebraic structure.

4.3.2.4 Initial Algebra Lemma

For $t, t' \in T_s(\Sigma_O, \Sigma_V)$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow t =_E t'.$$

Proof:

$\Rightarrow$: Immediately from definition 4.3.2.2.d.

$\Leftarrow$: For all contexts $T(\bullet_s)$ such that $T(t), T(t') \in T(\Sigma_O, \Sigma_V)$, it holds that $t =_E t'$, hence $T(t) =_E T(t')$ holds. □

4.3.2.5 Corollary ($\Sigma_O = \Sigma_V$ preserves initial algebra semantics)

For $t, t' \in T_s(\Sigma_V)$:

$$t \sim_{E,\Sigma_V} t' \Leftrightarrow t =_E t'.$$

4.3.2.6 Witness Existence Lemma

The following lemma formulates a nice fact for proofs with observable terms. Two terms are Σ_O-equal unless there is a context proving the opposite. Hence two terms are Σ_O-equal when there is no common context. So it is important to have at least one common context. In this lemma existence of a witness context is proven for Σ_O-observable terms of the same sort.

Lemma:
For two Σ_O-observable terms $t, t' \in T_{s,V}$ there exists a context $T(\bullet_s)$ such that

$T(t), T(t') \in T_O$.

Proof:

a. If $t, t' \in T_{s,O}$ the empty context $T(\bullet_s) = \bullet_s$ is fulfilling the condition.

b. If t is indirectly Σ_O-observable there exists a non-empty context $T(\bullet_s)$ such that $T(t) \in T_O$. Since T is non-empty the head function f is in F_O with range in S_O. Hence $T(t') \in T_O$.□

In the proof, case a corresponds to the initial algebra equality, and case b to the final algebra (observable only) equality.

4.3.2.7 Σ_O-equality as congruence: a problem with transitivity

We would have liked to use the notation of $=_O$ instead of $\sim_O$ since it should define a congruence similar to $=_E$. However, there are some problems connected with the final nature of $\sim_O$ and the initial nature of $=_E$. A congruence $\sim$ satisfies the following laws:

- *symmetry*, i.e., $t \sim t$;
- *reflexivity*, i.e., if $t \sim t'$ then also $t' \sim t$;
- *transitivity*, i.e., if $t \sim t'$ and $t' \sim t''$; then also $t \sim t''$;
- *the substitution property*, i.e., if $t_1 \sim t'_1 \wedge ... \wedge t_n \sim t'_n$ holds then also $f(t_1, ..., t_n) \sim f(t'_1, ..., t'_n)$ holds.

For terms in $T(\Sigma_V)$ reflexivity, symmetry and the substitution property of $\sim_O$ follow immediately from the corresponding properties of $=_E$ and definition 4.3.2.2.d. However, in section 4.3.2.3 facts d and e show that transitivity is not guaranteed on $T(\Sigma_V)$. Since these facts deal with terms that are not observable, this is no real problem. However, $\sim_O$ is also not transitive on the subset of Σ_O-observable terms in $T(\Sigma_V)$. This is illustrated in the following example.

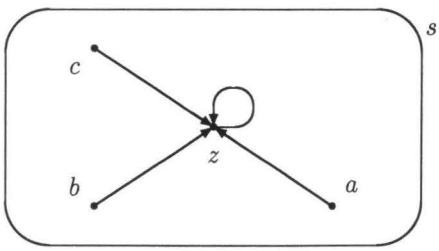


Figure 4.3.1

Let $\Sigma_V = (\{s\}, \{a, b, c, z, f\})$ with $a, b, c, z \in s$ and $f : s \rightarrow s$, E consists of the equation $f(x) = z$ with x a variable of sort s, and $\Sigma_O = (\{s\}, \{a, b, z, f\})$. This

structure is shown in Figure 4.3.1. Then $a \sim_O c$, since $f(a) =_E f(c)$, $f(f(a)) =_E f(f(c))$, etc., and similarly $c \sim_O b$, though $\neg a \sim_O b$.

Still, this is not unreasonable. If one only looks at Σ_O any relation involving c is irrelevant. The new structure is given in Figure 4.3.2, forgetting the dashed arrow.

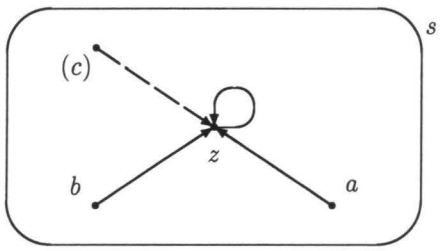


Figure 4.3.2

If later on one would want to add a new constant named c in Figure 4.3.2 then c could be a new name for an old constant like a, b, or z, or even a completely new constant. So if c is observably not equal to one or more of these constants, this would rule out some of the possibilities. Hence, the freedom allowed when introducing c would be limited in an undesirable way.

The precise criteria conserving transitivity, and hence making $\sim_O$ a congruence are given in Theorem 4.3.2.8. Some important classes of observable signatures that are transitivity conserving are given in a corollary (4.3.2.9).

4.3.2.8 Transitivity Theorem

Let t, t', t'' be Σ_O-observable terms of sort s such that $t \sim_O t'$ and $t' \sim_O t''$, then

$$\begin{aligned} \neg t \sim_O t'' \quad \Leftrightarrow \quad & t, t'' \in T_O \wedge t' \in T_V - T_O \wedge \\ & \forall \mathrm{T}(\bullet_s) : \mathrm{T}(t), \mathrm{T}(t'') \in T_O \rightarrow [\mathrm{T}(t) \neq_E \mathrm{T}(t'') \leftrightarrow \mathrm{T}(\bullet_s) = \bullet_s] \end{aligned}$$

Proof:

$\Leftarrow$: The empty context $\mathrm{T}(\bullet_s) = \bullet_s$ is a context for t and t'', since they are directly Σ_O-observable. Hence $t \neq_E t''$ and thus $\neg t \sim_O t''$.

$\Rightarrow$: The three parts of the conjunction are treated in sequence.

If t is indirectly Σ_O-observable all contexts $\mathrm{T}(\bullet_s)$ such that $\mathrm{T}(t), \mathrm{T}(t'') \in T_O$ are non-empty and hence satisfy $\mathrm{T}(t') \in T_O$. Hence for any such context $\mathrm{T}(t) =_E \mathrm{T}(t') =_E \mathrm{T}(t'')$, and thus $t \sim_O t''$. From this contradiction it follows that t (and by symmetry t'') is directly Σ_O-observable.

If t' is directly Σ_O-observable, $t =_E t'$ and $t' =_E t''$ hold, since t and t'' are directly observable, and hence $t \sim_O t''$. Hence t' must be indirectly Σ_O-observable.

Since $\neg t \sim_O t''$ there exists a context $\mathrm{T}(\bullet_s)$ such that $\mathrm{T}(t) \neq_E \mathrm{T}(t'')$ and $\mathrm{T}(t), \mathrm{T}(t'') \in T_O$. If $\mathrm{T}(\bullet_s)$ is non-empty then also $\mathrm{T}(t') \in T_O$, and hence $\mathrm{T}(t) =_E \mathrm{T}(t') =_E \mathrm{T}(t'')$. Thus $\mathrm{T}(\bullet_s)$ must be the empty context $\bullet_s$.□

4.3.2.9 Transitivity conserving constraints

The transitivity theorem states that two *directly* Σ_O-observable terms can be Σ_O-unequal, even though there is an *indirectly* Σ_O-observable term which is Σ_O-equal to both. This is the case in the example in 4.3.2.7. Hence Σ_O-inequality is stronger for directly observable terms than for indirectly observable terms.

The theorem above provides necessary and sufficient conditions for transitivity. This may be unwieldy to use in practice. However, it is conveniently possible to give criteria, that are important from the point of view of implementation, to check whether $\sim_O$ is an equivalence relation. Intuitively, the implementation of directly observable terms only has to follow the initial algebra semantics (Lemma 4.3.2.4), while indirectly observable and unobservable terms are less demanding for the implementation. The criteria are formulated below:

Corollary:

Let T_s be the subset of Σ_O-observable terms from $T_{s,V}$. The relation $\sim_{E,\Sigma_O}$ is an equivalence relation on T_s if one of the following holds:

a. $T_{s,O} = \emptyset$;

b. $T_{s,O} = T_s$;

c. for all $t \in T_s$ there is precisely one $t' \in T_{s,O}$ such that $t' \sim_O t$;

d. $\Sigma_O = (S_O, F_V)$ for some $S_O \subset S_V$.

Notes:

ad a. Sort s is not directly observable. Consequently its internal representation may be changed without altering the directly observable sorts.

ad b. All observable terms are directly observable, hence T_s has to be implemented with initial algebra semantics.

ad c. There is exactly one directly observable term Σ_O-equal to any term of T_s. This term plays the role of a canonical form and has to be implemented faithfully. All other terms may be implemented by their canonical equivalent.

ad d. If $s \in S_O$ then all constructor functions for terms of sort s are available, hence $T_s = T_{s,O}$ (case b holds for s).

If $s \notin S_O$ then no constructor function for terms of sort s qualifies as outermost function in T_O and hence $T_{s,O} = \emptyset$ (case a).

This case states that for s-observability $\sim_O$ is a congruence for terms of any sort $s' \in S_V$, including s itself. Hence it is a rephrasing of the well-known fact that observability through a sort conserves the congruence (see [MG85]).

In general $\sim_O$ will be a congruence. If that is the case it is usually written as $=_O$ in the sequel. Similarly $\sim_{E,\Sigma_O}$ becomes $=_{E,\Sigma_O}$ and $\sim_{E,\Sigma_V}$ becomes $=_{E,\Sigma_V}$.

The approaches in [BBK92] and [Hen91] allow for the definition of the same cases by carefully trimming the set of observing terms. Both however also allow for much wilder structures, since new observing terms may be added in a rather random way. While this may not make sense, it inhibits correctness proofs of the implementation.

4.3.2.10 Definition (Algebraic Implementation)

The following definition represents the central notion in this section, namely the notion of implementation for an algebraic specification relative to an observing signature. Intuitively, two specifications are algebraic implementations of each other when they have the same congruence on the observable terms. This is inherently an almost symmetric notion: if a small specification implements part of a large one then the large specification implements the same part of the small one (and more, but that is redundant) if the set of observable terms is the same. We provide the following *definition*:

- Let (Σ_V, Σ_H, E) and $(\Sigma'_V, \Sigma'_H, E')$ be algebraic specifications and Σ_O be a signature such that $\Sigma_O \subset \Sigma_V \cap \Sigma'_V$.

  $(\Sigma'_V, \Sigma'_H, E')$ is a Σ_O**-implementation of** (Σ_V, Σ_H, E) if and only if for all $s \in S_V$ and for all Σ_O-observable terms $t, t' \in T_s(\Sigma_V)$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow t \sim_{E',\Sigma_O} t'.$$

4.3.2.11 Some facts about algebraic implementations

a. If $\Sigma_O \subset \Sigma_V$ then an algebraic specification (Σ_V, Σ_H, E) is a Σ_O-implementation of itself. As an even more trivial special case (Σ_V, Σ_H, E) is a Σ_V-implementation of itself.

b. If $(\Sigma'_V, \Sigma'_H, E')$ is a Σ_O-implementation of (Σ_V, Σ_H, E) and $\Sigma'_O \subset \Sigma_O$ then $(\Sigma'_V, \Sigma'_H, E')$ is also a Σ'_O-implementation of (Σ_V, Σ_H, E).

c. Σ_O-implementation is a symmetric relation on the class of algebraic specifications with the same set of Σ_O-observable terms.

d. Σ_O-implementation is also a transitive relation under the conditions of case c.

While the facts above provide some idea about the usefulness of the definition two important properties have to be proved. Of course we want to conserve the property in initial algebra semantics that the hidden signature and the set of equations may be changed as long as this does not affect the congruence on the visible signature. This is proved in lemma 4.3.2.12.

Next, in the central theorem a functionally oriented criterion is given for an algebraic implementation. This serves as a starting point for section 4.5, in which a notion of functional implementation will be given.

4.3.2.12 Initial Algebra Implementation Lemma

Let $(\Sigma_V, \Sigma'_H, \mathrm{E}')$ be a Σ_V-implementation of $(\Sigma_V, \Sigma_H, \mathrm{E})$, then for all $s \in \mathrm{S}_V$ and for all $t, t' \in T_{s,V}$:

$$t =_{\mathrm{E}} t' \Leftrightarrow t =_{\mathrm{E}'} t'.$$

Proof:
All terms in T_V are Σ_V-observable. Hence for all $s \in \mathrm{S}_V$ and for all $t, t' \in T_{s,V}$:

$$t =_{\mathrm{E},\Sigma_V} t' \Leftrightarrow t =_{\mathrm{E}',\Sigma_V} t'.$$

According to corollary 4.3.2.5 $t =_{\mathrm{E},\Sigma_V} t' \Leftrightarrow t =_{\mathrm{E}} t'$ and $t =_{\mathrm{E}',\Sigma_V} t' \Leftrightarrow t =_{\mathrm{E}'} t'$, hence $t =_{\mathrm{E}} t' \Leftrightarrow t =_{\mathrm{E}'} t'$.□

4.3.2.13 Algebraic Implementation Theorem

Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ and $(\Sigma'_V, \Sigma'_H, \mathrm{E}')$ be algebraic specifications and $\Sigma_O \subset \Sigma_V \cap \Sigma'_V$.

If for all $f \in \mathrm{F}_O$, $f : s_1 \times ... \times s_k \rightarrow s_0$, with $s_0 \in \mathrm{S}_O$, for all $t \in T_{s_0}(\Sigma_V)$ and for all $(u_1, ..., u_k) \in (T(\Sigma_V))^k$ $f(u_1, ..., u_k) =_{\mathrm{E}} t \Leftrightarrow f(u_1, ..., u_k) =_{\mathrm{E}'} t$ holds, then $(\Sigma'_V, \Sigma'_H, \mathrm{E}')$ is a Σ_O-implementation of $(\Sigma_V, \Sigma_H, \mathrm{E})$.

Proof:
Let $s \in \mathrm{S}_V$ and $t, t' \in T_s(\Sigma_V)$ be Σ_O-observable, then

$$\begin{aligned}
& t \sim_{\mathrm{E},\Sigma_O} t' \\
\Leftrightarrow\ & \forall \mathrm{T}(\bullet_s)[\mathrm{T}(t), \mathrm{T}(t') \in T(\Sigma_O, \Sigma_V) \rightarrow \mathrm{T}(t) =_{\mathrm{E}} \mathrm{T}(t')] && \text{(4.3.2.2.d)} \\
\Leftrightarrow\ & \forall \mathrm{T}(\bullet_s)\mathrm{T}(t), \mathrm{T}(t') \in T(\Sigma_O, \Sigma_V)[\exists g \in \mathrm{F}_O, u_1, ..., u_k \in T(\Sigma_V) && \text{(4.3.2.2.a)} \\
& (g(u_1, ..., u_k) = \mathrm{T}(t) \wedge g(u_1, ..., u_k) =_{\mathrm{E}} \mathrm{T}(t'))] \\
\Leftrightarrow\ & \forall \mathrm{T}(\bullet_s)\mathrm{T}(t), \mathrm{T}(t') \in T(\Sigma_O, \Sigma_V)[\exists g \in \mathrm{F}_O, u_1, ..., u_k \in T(\Sigma_V) \\
& (g(u_1, ..., u_k) = \mathrm{T}(t) \wedge g(u_1, ..., u_k) =_{\mathrm{E}'} \mathrm{T}(t'))] \\
\Leftrightarrow\ & \forall \mathrm{T}(\bullet_s)\mathrm{T}(t), [\mathrm{T}(t') \in T(\Sigma_O, \Sigma_V) \rightarrow \mathrm{T}(t) =_{\mathrm{E}'} \mathrm{T}(t')] && \text{(4.3.2.2.a)} \\
\Leftrightarrow\ & t \sim_{\mathrm{E}',\Sigma_O} t'. && \text{(4.3.2.2.d)}
\end{aligned}$$

□

Note: this theorem is sufficiently strong to describe the behaviour of a function up to the congruence defined by $\sim_{\mathrm{E},\Sigma_O}$, if such a congruence exists. An example of the use of the theorem is given in the next section. A more restrictive definition of implementation, strong enough to describe functional implementation, is given in section 4.5.

4.4 An example: Tables

In this section, two definitions of elementary data structures for objects of arbitrary sort ELEM are given. Both data structures support storage and retrieval with elements of an arbitrary sort KEY as selection criterion. Equality is defined on this sort through function eq.

The first specification describes a sort TABLE. An element of this sort is a list of all entries with corresponding keys in the data structure. This data structure can be searched from the last entry to the first, but anything faster than linear search is not accommodated for.

The second specification uses a hidden sort TREE to implement the same data in a search-tree. This is possible when we have a total ordering on sort KEY. In the example function lt, combined with eq, provides such an ordering.

The specifications below are parameterized with sorts KEY and ELEM at the specification level. To guide the intuition, one should think of two (perhaps equal) sorts which are already well-known, e.g. CHAR for ELEM and INT for KEY. ASF does not provide a semantics for unbound parameter sorts, since there is no mechanism to force restrictions on the actual parameter (like the total ordering in the example). In this Chapter restrictions are given in the commentary, so it will be clear what the semantics should be.

The remainder of this section is devoted to a fairly detailed proof sketch of the fact that the modules Tables and Tables-as-trees are algebraic implementations of each other when one observes through the retrieve function lookup only, in other words, with respect to lookup-equality. So we take for the observing signature ({BOOL, ELEM, KEY, TABLE}, {lookup}), or, according to fact 3.2.3.g, ({ELEM}, {lookup}), since ELEM is the range of lookup.

The simple specification is given below:

```
module Tables -- original specification
begin
  parameters Keys-and-Elements
    begin
      sorts KEY, ELEM
      functions eq: KEY # KEY -> BOOL -- equality
    end Keys-and-Elements
  exports
    begin
      sorts TABLE
      functions
        nulltable:                    -> TABLE
        tableadd : KEY # ELEM # TABLE -> TABLE
        lookup   : KEY # TABLE        -> ELEM
        errorelem:                    -> ELEM
    end
  imports Booleans
```

```
  variables key, key1, key2: -> KEY
            elem           : -> ELEM
            table          : -> TABLE
  equations
    [1] lookup(key, nulltable) = errorelem
    [2] lookup(key1, tableadd(key2,elem,table))
            = if(eq(key1,key2),
                 elem,
                 lookup(key1,table))
end Tables
```

This specification speaks for itself. It is similar to the first `BoundedArray` specification in section 4.3.1.

Function `tableadd` gives the same problems as function `put` in that module. To avoid them we restrict the set of observing terms to those with function `lookup` as outermost symbol. Hence an implementor of this module can concentrate on the correct implementation of `lookup`.

To get an efficient implementation of `lookup` more detailed information about sort `KEY` is needed. If `KEY` is a small set something similar to a bounded array is feasible. If a hash function could be defined, a hash table might be used as implementation. Each of these structures can be algebraically specified as hidden structure, thus providing an algebraic specification which upon implementation gives an equivalent, but more efficient, implementation of `lookup`.

For this example it is assumed that a total ordering can be defined on the set `KEY` with the functions `eq` and `lt` (lower-than). The total ordering allows the definition of a binary search-tree. This is done in module `Tables-as-trees` below:

```
module Tables-as-trees
begin
  parameters Keys-and-Elements
    begin
      sorts KEY, ELEM
      functions eq: KEY # KEY -> BOOL -- equality
                lt: KEY # KEY -> BOOL -- lower-than -- new
      -- eq and lt must provide a total ordering on sort KEY
    end Keys-and-Elements

  exports
    begin
      sorts TABLE
      functions
        nulltable:                    -> TABLE
        tableadd : KEY # ELEM # TABLE -> TABLE
        lookup   : KEY # TABLE        -> ELEM
        errorelem:                    -> ELEM
    end
```

```
  imports Booleans

  -- hidden section
  sorts TREE                                          -- new
  functions
    tree     : TREE # KEY # ELEM # TREE ->  TREE  -- new
    niltree  :                          ->  TREE  -- new
    treeadd  : KEY # ELEM # TREE        ->  TREE  -- new
    lookuptr : KEY # TREE               ->  ELEM  -- new
    tbltotree: TABLE                    ->  TREE  -- new

  variables key, key2   : -> KEY
            elem, elem2 : -> ELEM
            table       : -> TABLE
            tree1, tree2: -> TREE
  equations                                           -- new
   [h1] tbltotree(nulltable) = niltree
   [h2] tbltotree(tableadd(key,elem,table))
                = treeadd(key,elem,tbltotree(table))

   [h3] treeadd(key,elem,niltree)
                = tree(niltree,key,elem,niltree)
   [h4] treeadd(key,elem,tree(tree1,key2,elem2,tree2))
                = if(eq(key,key2),
                     tree(tree1,key,elem,tree2),
                     if(lt(key,key2),
                        tree(treeadd(key,elem,tree1),
                             key2,elem2,tree2),
                        tree(tree1,key2,elem2,
                             treeadd(key,elem,tree2))))

   [h5] lookuptr(key,niltree) = errorelem
   [h6] lookuptr(key,tree(tree1,key2,elem,tree2))
                = if(eq(key,key2),
                     elem,
                     if(lt(key,key2),
                        lookuptr(key,tree1),
                        lookuptr(key,tree2)))
   [h7] lookup(key,table) = lookuptr(key,tbltotree(table))
end Tables-as-trees
```

Note that all equations contain hidden sorts. Equation h7 defines `lookup` in terms of `lookuptr`, the retrieval function on trees, itself defined in h5 and h6. Equations h1 through h4 define the build-up of a tree from a table.

It is possible to declare all hidden sorts and functions visible rather than hidden. The effect would be that module `Tables-as-trees` would still be an implementa-

tion with respect to `lookup`-observability of module `Tables`, but not the other way around. The reason for the latter is the existence of observable terms containing constructor functions for `TREE` in module `Tables-as-trees`, terms which are not existent in module `Tables`.

The following proof sketch first defines a well-formedness predicate *searchtree* for terms of sort `TREE`, since not all constructible terms are search-trees. Then it is proved that the predicate *searchtree* is invariant over the insertion function `treeadd`, and that the retrieval function `lookuptr` is well-defined for single additions to a tree which satisfies this predicate. Finally the equivalence between the two specifications is proved by induction on the number of insertions.

4.4.1 Definition (well-formedness of search trees)

The predicate *searchtree*(`t`) for a term `t` of sort `TREE` describes the well-formedness of a tree as search-tree. It will be used in the proof to derive properties about the behaviour of the data structure generated by function `treeadd`. This holds in particular for the behaviour observed through function `lookuptr`, which is needed to derive the behaviour of function `lookup`. The predicate is defined as follows (with `t1`, `t2` of sort `TREE`, `j`, `k`, `l` of sort `KEY`, and `e` of sort `ENTRY`):

- *searchtree*(`niltree`) = true;
- *searchtree*(`tree(t1,k,e,t2)`) =
  searchtree(`t1`) $\wedge$ *searchtree*(`t2`) $\wedge$
  $\forall$ `j` $\in$ *set-of-keys*(`t1`) [`lt(j,k) = true`] $\wedge$
  $\forall$ `l` $\in$ *set-of-keys*(`t2`) [`lt(k,l) = true`],

with *set-of-keys*(`t`) for terms `t` of sort `TREE` a set containing all keys in `t`. Formally:

- *set-of-keys*(`niltree`) = $\emptyset$;
- *set-of-keys*(`tree(t1,k,e,t2)`)
  = *set-of-keys*(`t1`) $\cup$ {`k`} $\cup$ *set-of-keys*(`t2`).

4.4.2 Well-formedness lemma for trees

Two important properties of the behaviour of well-formed trees (i.e., terms satisfying predicate *searchtree*) are formulated. Case (a) states that the predicate *searchtree* is an invariant under insertion in the tree. Case (b) states that a well-formed tree behaves properly with respect to function `lookuptr` after insertion. These facts provide technical steps for the proof of `lookup`-equivalence in section 4.4.3.

For the remainder of section 4.4, let `k`, `k'` be of sort `KEY`, `e`, `e'` be of sort `ENTRY`, and `t`, `t'` be of sort `TREE`, then we can formulate the following

Lemma:

a. *searchtree*(`t`) $\rightarrow$ *searchtree*(`treeadd(k',e',t)`).

b. *searchtree*(t) $\rightarrow$

[eq(k,k') = true $\rightarrow$ lookuptr(k,treeadd(k',e',t)) = e'] $\wedge$

[eq(k,k') = false

$\rightarrow$ lookuptr(k,treeadd(k',e',t)) = lookuptr(k,t)].

Proof by induction on the number of nodes in the tree (omitted).

4.4.3 Proof of lookup-equality

The following proof uses induction with respect to the number of insertions using function tableadd. The well-formedness predicate *searchtree* makes the proof straightforward.

In this proof the equivalence defined by the equations from module Tables is called $=_{\mathrm{Tb}}$ and from module Tables-as-trees $=_{\mathrm{Tr}}$. Equivalence according to an equation numbered i is written $=_i$.

According to Theorem 4.3.2.13 it is sufficient to prove for all pairs (k, t) $\in T_{\mathbf{KEY},V}$ $\times$ $T_{\mathbf{TABLE},V}$ and for all terms e $\in T_{\mathbf{ELEM},V}$

$$\texttt{lookup(k,t)} =_{\mathrm{Tb}} \texttt{e} \Leftrightarrow \texttt{lookup(k,t)} =_{\mathrm{Tr}} \texttt{e}.$$

First we assume that e does not contain the function lookup. The proof then proceeds with induction on the length of terms in $T_{\mathbf{TABLE},V}$, which is defined in the obvious way, with multiple occurrences of the same key counted separately for every occurrence.

For the table of length 0, nulltable, both lookup(k, nulltable) $=_{\mathrm{Tb}}$ errorelem and lookup(k, nulltable) $=_{\mathrm{Tr}}$ errorelem obviously hold. Now let

$$\texttt{lookup(k,t)} =_{\mathrm{Tb}} \texttt{e} \Leftrightarrow \texttt{lookup(k,t)} =_{\mathrm{Tr}} \texttt{e}$$

be proved for all tables of length $n \geq 0$ and e not containing lookup, and let t be a table of length n, hence t' = tableadd(k',e',t), with e' not containing function lookup, is a table of length $n + 1$.

- If eq(k,k') $=_{\mathrm{Tb}}$ true (and hence eq(k,k') $=_{\mathrm{Tr}}$ true) then

  lookup(k,t') $=_2$ e', and

  lookup(k,t') $=_{\mathrm{h7}}$ lookuptr(k,tbltotree(tableadd(k',e',t)))

  $=_{\mathrm{h2}}$ lookuptr(k,treeadd(k',e',tbltotree(t)))

  $=_{\mathrm{Tr}}$ e',

  with the last equation following from lemma 4.4.2.

- If eq(k,k') $=_{\mathrm{Tb}}$ false (hence also eq(k,k') $=_{\mathrm{Tr}}$ false) then

  lookup(k,t') $=_2$ lookup(k,t), and

  lookup(k,t') $=_{\mathrm{h7,h2}}$ lookuptr(k,treeadd(k',e',tbltotree(t)))

  $=_{\mathrm{Tr}}$ lookuptr(k,tbltotree(t))

  according to lemma 4.4.2. The induction hypothesis states that lookuptr(k, tbltotree(t)) $=_{\mathrm{Tr}}$ lookup(k, t).

The proof can now be extended to general terms $\mathtt{e} \in T_{\mathtt{ELEM},V}$ by replacing such terms by terms not containing function `lookup`, starting with the innermost occurrence(s) of this function. It can easily be seen that any term in $T_{\mathtt{ELEM},V}$ containing one occurrence of `lookup` is equivalent in either module to a term containing no occurrence of `lookup`. The soundness of such a replacement per term with one `lookup` was proved above. Since `e` contains a finite number of occurrences of this function this series of replacements terminates.

Hence the proof sketch is complete for general $\mathtt{e} \in T_{\mathtt{ELEM},V}$.

4.5 Functional implementation

4.5.1 The functional view

The implementation Theorem (4.3.2.13 in section 4.3.2) gives an algebraically clean criterion for implementation. However, it is not sufficient as a tool to fix implementations of functions in the classical sense: a function has a certain result value for every combination of input values. Of course the result value should depend on the input values, but it should not depend on the implementation.

A violation of this property is shown in the example below. Let:
$\Sigma_V = (\{s, t\}, \{a, b, p, q, \mathrm{f}\})$ with $a, b \in s$, $p, q \in t$ and f: $s \rightarrow t$,
$\Sigma_O = (\{s, t\}, \{a, b, \mathrm{f}\})$,
$\Sigma_H = \emptyset$,
$\mathrm{E} = \{\mathrm{f}(a) = p, \mathrm{f}(b) = p\}$, and
$\mathrm{E}' = \{\mathrm{f}(a) = q, \mathrm{f}(b) = q\}$.

The Σ_O-observable terms in T_V are a, b, f(a) and f(b). Obviously $\mathrm{f}(a) =_{\mathrm{E},\Sigma_O} \mathrm{f}(b)$ and $\mathrm{f}(a) =_{\mathrm{E}',\Sigma_O} \mathrm{f}(b)$. Hence $(\Sigma_V, \emptyset, \mathrm{E})$ and $(\Sigma_V, \emptyset, \mathrm{E}')$ are Σ_O-implementations of each other. However, f clearly has different result values, unless $p = q$.

Additional restrictions are needed to be able to view a term in T_O as a function (the header function) defined on tuples in T_V and with range T_V. In initial algebra semantics the 'result' is the congruence class defined by the set of equations E. Hence any term in the congruence class will do, since it fixes (for specific E and Σ_V) the class. So we need a canonical form, which is a representative for every congruence class. In a confluent and terminating term rewriting system this canonical form is called 'normal form', and it is defined by the system itself.

The following three sets of terms within T_V are induced by Σ_O:

- the *directly* Σ_O-observable terms,
- the *indirectly* Σ_O-observable terms, and
- the terms *reachable* from T_O, i.e., terms not necessarily in T_O but in the congruence class of some term in T_O.

Note that the last two sets may overlap. The input values for functions in F_O with range in S_O form a subset of the union of the first two sets. Any element of the first and the third sets could be in the range of a function in F_O.

The directly Σ_O-observable terms do not necessarily contain a desired result value. For example, a specification of *string-of-characters* might contain a function *length* from strings to integers. The set of *length*-observable terms contains the *length* function applied to numerous strings of various length, but it does not contain the integers, which is clearly the desired set of result values.

In the subsection below this idea is formalized for a specific observing function. The function has input terms, which should be well-typed, and an output term, depending on the input terms and the set of equations, which must be in a certain set of canonical terms. There is an obvious link with the theory of *abstract data types* (cf. Jones [Jon80]) here. The well-typedness of the input terms serves as

precondition and the equations and a characterization of the set of canonical terms serve as postcondition.

In general one has more than one observing function, so some preliminary work has to be done to allow a decomposition of the observing set of functions into singletons.

4.5.2 A theory of functional implementation

4.5.2.1 Definitions (input-, reachable and canonical terms)

Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ be an algebraic specification and $\Sigma_O \subset \Sigma_V$. Then

a. the set $I(\Sigma_O, \Sigma_V)$ of Σ_O**-input terms over** Σ_V is defined as:

$$I(\Sigma_O, \Sigma_V) =$$

$$\{t \in T(\Sigma_V) \mid \exists f \in \mathrm{F}_O,\ s \in \mathrm{S}_O,\ f : s_1 \times \ldots \times s_k \rightarrow s, \exists i \leq k\, [t \in T_{s_i}(\Sigma_V)]\}.$$

b. the set $R(\Sigma_O, \Sigma_V)$ of Σ_O**-reachable terms over** Σ_V is defined as:

$$R(\Sigma_O, \Sigma_V) = \{t \in T(\Sigma_V) \mid \exists t' \in T(\Sigma_O, \Sigma_V)\, t =_{\mathrm{E}} t'\}.$$

Note that terms containing hidden functions and sorts are not considered reachable.

c. A set $C(\Sigma_O, \Sigma_V) \subset R(\Sigma_O, \Sigma_V)$ is a **set of canonical terms** if and only if

$$\forall t, t' \in C(\Sigma_O, \Sigma_V)[t =_{\mathrm{E}} t' \rightarrow t = t']$$

with $=$ syntactic equality.

d. A set of canonical terms $C(\Sigma_O, \Sigma_V)$ is **complete** if and only if

$$\forall t \in T(\Sigma_O, \Sigma_V)\ \exists\, t' \in C(\Sigma_O, \Sigma_V)\, t =_{\mathrm{E}} t'.$$

e. A **reduction to canonical terms** $\twoheadrightarrow_{C(\Sigma_O, \Sigma_V)}$ (abbreviated $\twoheadrightarrow_C$) is defined as follows:

$$t \twoheadrightarrow_{C(\Sigma_O, \Sigma_V)} t' \Leftrightarrow t \in R(\Sigma_O, \Sigma_V) \wedge t' \in C(\Sigma_O, \Sigma_V) \wedge t =_{\mathrm{E}} t'.$$

f. Analogous to the definitions of T_O and $T_s(\Sigma_O, \Sigma_V)$ the following shorthand conventions are adopted:

$$\begin{array}{lcll} I_O & = & I(\Sigma_O, \Sigma_V), & \\ R_O & = & R(\Sigma_O, \Sigma_V), & \\ C_O & = & C(\Sigma_O, \Sigma_V), & \\ I_{s,O} & = & I_s(\Sigma_O, \Sigma_V) & = I(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V), \\ R_{s,O} & = & R_s(\Sigma_O, \Sigma_V) & = R(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V), \text{ and} \\ C_{s,O} & = & C_s(\Sigma_O, \Sigma_V) & = C(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V). \end{array}$$

4.5.2.2 Some facts

a. $R(\Sigma_O, \Sigma_V) \supseteq T(\Sigma_O, \Sigma_V)$.

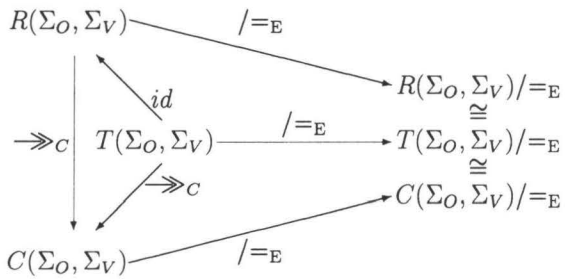
b. Every term in $I(\Sigma_O, \Sigma_V)$ is Σ_O-observable, provided that $\forall s \in S_V\, T_s(\Sigma_V) \neq \emptyset$.

c. The converse of fact b does not hold, i.e., not every Σ_O-observable term is a Σ_O-input term.

4.5.2.3 Lemma

Let $C(\Sigma_O, \Sigma_V)$ be a complete set of canonical terms. The operation of the functions on $R(\Sigma_O, \Sigma_V)$ can be restricted to reach $C(\Sigma_O, \Sigma_V)$ by application of the reduction to canonical terms $\twoheadrightarrow_C$ after the normal application of the function in $R(\Sigma_O, \Sigma_V)$, so that the terms in $C(\Sigma_O, \Sigma_V)$ with these functions form a term algebra. Then $C(\Sigma_O, \Sigma_V)$ as a term algebra is isomorphic to $R(\Sigma_O, \Sigma_V)/=_E$.

Proof (sketch):
Since $C(\Sigma_O, \Sigma_V)$ is complete the following diagram commutes:



By definition $C(\Sigma_O, \Sigma_V) \cong C(\Sigma_O, \Sigma_V)/=_E$ and it hence follows that $C(\Sigma_O, \Sigma_V) \cong R(\Sigma_O, \Sigma_V)/=_E$.□

4.5.2.4 Functional decomposition of a reduction to canonical terms

Next we want to pursue a 'divide and conquer' strategy to provide an implementation of a reduction to canonical terms $\twoheadrightarrow_C$. The decomposition chosen is made on the typed head symbol (the "function") of the term to be reduced. This allows for a separate implementation for each function. The total implementation of $\twoheadrightarrow_C$ can be constructed from the union of these separate implementations.

It should be noted that a reduction to canonical terms $\twoheadrightarrow_C$ as a total map from R_O to C_O is fixed by C_O and the congruence $/=_E$. This follows from the definition of C_O, since for every term in R_O exactly one term in C_O is in the same congruence class. So it is possible to define the map $\twoheadrightarrow_C$ as a union of partial maps to the set of canonical terms.

It is also possible to define a complete set of canonical terms implicitly by defining a (possibly partial) map $\twoheadrightarrow$ from T_O to R_O for which the following holds:

1) $\forall t \in T_O, t' \in R_O[t \twoheadrightarrow t' \Leftrightarrow t =_{\mathrm{E}} t']$
2) $\forall t \in T_O \; card(\{t' \in R_O \mid \exists t'' \in T_O[t =_{\mathrm{E}} t'' \wedge t'' \twoheadrightarrow t']\}) = 1.$

Obviously the range of $\twoheadrightarrow$ is a complete set of canonical terms. So a reduction to canonical terms can be described by its behaviour on terms in T_O. A well-known example of such an implicit definition is the set of normal forms defined by a confluent and terminating term rewriting system.

4.5.2.5 Definitions (functional implementation)

a. Let $\twoheadrightarrow_C$ be a reduction to canonical terms and let $\Sigma \subset \Sigma_O$. Then $\twoheadrightarrow_{\Sigma,C}$ is defined as the restriction of $\twoheadrightarrow_C$ to the domain $T(\Sigma, \Sigma_V)$.

b. Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ be an algebraic specification, Σ_O be a signature such that $\Sigma_O \subset \Sigma_V$, and C_O be a complete set of canonical terms. A map $\twoheadrightarrow$ is a **functional implementation** if and only if

$$\forall t \in T_O, t' \in C_O[t \twoheadrightarrow t' \leftrightarrow t =_{\mathrm{E}} t'].$$

The notation $\twoheadrightarrow$ for the map intuitively resembles the notation of a reduction to canonical terms. Of course, if the map is not a functional implementation, a plain functional notation might be more appropriate.

4.5.2.6 Some facts about functional implementations

a. Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ be an algebraic specification, $\Sigma_O \subset \Sigma_V$, C_O a set of canonical terms, and $\twoheadrightarrow_C |_{T_O}$ the restriction of reduction to canonical terms $\twoheadrightarrow_C$ to domain T_O. Then:

$$\twoheadrightarrow_C |_{T_O} = \bigcup_{f \in \mathrm{F}_O} \twoheadrightarrow_{(\mathrm{S}_O, \{f\}), C}.$$

Hence $\twoheadrightarrow_C |_{T_O}$ — and according to section 4.5.2.4 thus by extension $\twoheadrightarrow_C$ — can be defined for each function in Σ_O separately.

b. Let $(\Sigma_V, \Sigma_H, \mathrm{E})$ and $(\Sigma'_V, \Sigma'_H, \mathrm{E}')$ be Σ_O-implementations of each other (so $\Sigma_O \subset \Sigma_V \cap \Sigma'_V$). Then a functional implementation $\twoheadrightarrow_C \subset T_O \times C_O$ of $(\Sigma'_V, \Sigma'_H, \mathrm{E}')$ is also a functional implementation of $(\Sigma_V, \Sigma_H, \mathrm{E})$ if for all $t \in T_O$, $t' \in C_O$ $t =_{\mathrm{E}} t' \leftrightarrow t =_{\mathrm{E}'} t'$ holds.

c. If two algebraic specifications are Σ_O-implementations of each other and both have a functional implementation then these implementations are isomorphic.

4.5.2.7 Concrete representation

Eventually, we want to convert an algebraic specification into a working computer program. For this a representation function ι from the set of input terms I_O to the concrete representation of input terms is needed to be able to execute implemented functions. When confusion arises the restriction of ι to the domain $I_{s,O}$ will be written as ι_s. Additionally, a set of retrieval functions ρ_s from concrete representations of output terms to the set of canonical terms $C_{s,O}$ is needed.

This is formalized in the following

Definitions:

Let I be a set of data types for a programming language L. I is an **implementation in L of I_O and R_O** if there is a total function ι: $I_O \rightarrow$I (the **implementation function**) and a set of (partial) functions $\{\rho_s$: I$\rightarrow I_{s,O} \cup R_{s,O} \mid s \in S_O\}$ (the **retrieval functions**) such that $\rho_s(\iota(t)) =_O t$ for all $t \in I_{s,O}$.

In general, if $I_{s,O}$ is not empty then $\iota(I_{s,O})$ will correspond to a subset of a data type in L. It could happen very well that two different input types are implemented by the same data type, so differently named retrieval functions are needed for every sort. Only one name (ι) is needed for the implementation function, since the sort of the argument provides typing information.

4.5.2.8 Implementation theorem

Let (Σ_V, Σ_H, E) be an algebraic specification, $\Sigma_O \subset \Sigma_V$, and C_O a set of canonical terms. Let I be an implementation in a programming language L of I_O and R_O with implementation function ι and retrieval functions $\{\rho_s \mid s \in S_O\}$, and $S(x_1, ..., x_n, r)$, a program operating on the implementation data I, which returns its result in variable r. Then the statement $S(x_1, ..., x_n, r)$, describes a functional implementation $\twoheadrightarrow_{(S_O,\{f\}),C}$ for $f : s_1 \times ... \times s_n \rightarrow s$, $s \in S_O$, if the following holds:

$$\{c_1 \in s_1 \wedge ... \wedge c_n \in s_n \wedge k_1 = \iota(c_1) \wedge ... \wedge k_n = \iota(c_n)\}$$
$$S(k_1, ..., k_n, r)$$
$$\{\rho_s(c) \in C_{s,O} \wedge f(c_1, ..., c_n) =_E \rho_s(r)\}.$$

Proof: Let function F: $s_1 \times ... \times s_n \rightarrow s$ be defined by $F(a_1, ..., a_n) = a$ if and only if $\rho_s(r) = a$ after execution of $S(\iota(a_1), ..., \iota(a_n), r)$, i.e., F is the function defined by S. Then $f(a_1, ..., a_n) =_E F(a_1, ..., a_n)$ and $F(a_1, ..., a_n) \in C_{s,O}$. Hence $f(a_1, ..., a_n)$ $\twoheadrightarrow_{(S_O,\{f\}),C} F(a_1, ..., a_n)$ holds.□

4.5.2.9 Decidability of the conditions

It is a pleasant property of Theorem 4.5.2.8 that in practice satisfaction of the precondition can be computed if the implementation function ι can be computed. Since the terms in I_O are typed, a typechecking algorithm provides the statements on membership of the input terms. In general there are no extra restrictions to ensure computability, since obviously the implementation has to be computed anyway.

The decidability of the postcondition depends on the computability of the retrieval function ρ_s, the decidability of the check on membership of the set of canonical

terms C_O, and the decidability of the congruence $=_E$. The first condition is necessarily fulfilled for the same reasons as the computability of the implementation function. The second depends on the definition of C_O, which will allow computation in practical cases (who wants a canonical form wild enough to be unrecognizable as such?). The decidability of $=_E$ is not ensured in general. So a separate proof may be needed. Of course, for many classes this congruence is decidable. For specifications where the congruence is undecidable, e.g. an algebraic specification of a programming language, an implementation will have to provide at least a partial decision procedure, even when it cannot be completed.

4.6 An example: Tables revisited

To illustrate the use of Theorem 4.5.2.8 an implementation of module `Tables` in an imperative language is given. Though the implementation again uses trees there is an important difference with the algebraic implementation in section 4.4 in the sense that recursion is eliminated.

The language Pascal (described in, e.g., [JW78]) is chosen for the imperative implementation. This choice is motivated by its availability and by its convenient type system. Of course, any other imperative language would serve as well. It should be noted that a functional implementation is very well possible, even in Pascal, but we want to illustrate the possibility to give a correct implementation in a non-functional way.

In general, it is easier to derive a functional program from an ASF-specification, since writing an algebraic specification has strong similarities to functional programming. The specification of `Tables-as-trees`, for instance, is easily converted into a functional program for `lookup`. Thus a functional implementation has the advantage of being easily derived from the specification, and also of being faster in general than a term rewriting implementation.

The first step in the implementation is the choice of a data structure. This is provided for by the following data type declarations:

```
type key = integer;
     elem = char;
     pointer = ↑tree;
     tree = record
                 l,r: pointer;
                 k: key;
                 e: elem
            end;
```

In a concrete program it is necessary to bind the sorts *key* and *elem*. The choice for integers and characters is arbitrary; the only prerequisite is that an ordering must be established on the keys. A node in a tree has four fields, a left and right pointer to subtrees, and information fields for key and element.

The values `niltree` and `errorelem` require different treatment in Pascal. For the first we can use the standard notion *nil*, the second has to be declared as variable and set to some unused value.

The auxiliary functions on *key* pose no problems with the current choice, since integers are already ordered, though of course another choice could make implementation much more complicated:

```
function eq (a,b: key): boolean;
begin eq := (a=b) end ;

function lt (a,b: key): boolean;
begin lt := (a < b) end ;
```

Next the implementation function ι must be defined. The domain of ι is $T_{\mathtt{KEY},V} \cup T_{\mathtt{ELEM},V} \cup T_{\mathtt{TABLE},V}$ and its range is the union of the data types *key*, *elem* and *tree* (or rather *pointer* to *tree*) already indicated above. Since a specification of the terms of type ELEM and KEY has not been given in section 4.3 an identification with *elem* and *key* is assumed, so ι is 'defined' backwards by $\iota(t) = t$ for $t \in T_{\mathtt{KEY},V} \cup T_{\mathtt{ELEM},V}$. Hence also $\rho_{\mathtt{ELEM}}(t) = t$, the only retrieval function needed for the example. For $t \in T_{\mathtt{TABLE},V}$ a definition of ι can be provided as follows:

ι(**nulltable**) = *nil*
ι(**tableadd(key,elem,table)**) = *ptr*
when *treeadd(*ι(**key**)*,* ι(**elem**) *,ptr)*
is executed with *ptr* = ι(**table**).

This definition uses procedure *treeadd* defined below. It should be noted that function ι restricted to terms of type TABLE plays the same role as function **tbltotree** in section 4.4. Evidently, procedure *treeadd* below and function **treeadd** in section 4.4 are closely related also. A procedure with a variable parameter is a common way to handle data structures in a language like Pascal. A function definition would have the advantage of a more elegant definition of function ι, but the definition below shows that other programming styles can be handled too.

```
procedure treeadd (ky: key; el: elem; var root: pointer);
var   cur, anc: pointer;
      inserted: boolean;
begin
      cur:= root;
      inserted:= false;
      while not inserted do
      begin
            if    cur = nil
            then
                  begin
                        new(cur);
                        cur↑.l:= nil; cur↑.r:= nil;
                        cur↑.k:= ky; cur↑.e:= el;
                        if root = nil
                        then root:=cur
                        else
                              if lt(ky, anc↑.k)
                              then anc↑.l:=cur
                              else anc↑.r:=cur;
                        inserted:= true
                  end
            else
                  begin
                        if eq(ky,cur↑.k)
```

```
                then begin cur↑.e:= el; inserted:= true end
                else
                        if lt(ky,cur↑.k)
                        then begin anc:=cur; cur:=cur↑.l end
                        else begin anc:=cur; cur:=cur↑.r end
            end
    end
end;
```

The proof of correctness of this implementation closely resembles the proof sketch in section 4.4. Hence it will be an even more concise sketch. Following the lead in section 4.4.1 we provide two well-formedness predicates on structures of type *pointer* (to *tree*), again called *searchtree* and *set-of-keys*. They are defined as follows (*ptr* of type *pointer* to *tree* and j, k, l, resp. e, of type *key*, resp. *entry*):

- $searchtree(nil) = \text{true}$;
- $searchtree(ptr) = searchtree(ptr\uparrow.l) \wedge searchtree(ptr\uparrow.r) \wedge$
  $\forall\, j \in set\text{-}of\text{-}keys(ptr\uparrow.l)\ [lt(j,ptr\uparrow.k) = true] \wedge$
  $\forall\, l \in set\text{-}of\text{-}keys(ptr\uparrow.r)\ [lt(ptr\uparrow.k,l) = true]$;

and

- $set\text{-}of\text{-}keys(nil) = \emptyset$;
- $set\text{-}of\text{-}keys(ptr) = set\text{-}of\text{-}keys(ptr\uparrow.l) \cup \{\ ptr\uparrow.k\ \} \cup set\text{-}of\text{-}keys(ptr\uparrow.r)$.

This allows us to state the well-formedness of implemented terms by providing the following parallel to Lemma 4.4.2.a:

4.6.1 Second well-formedness lemma for trees (part a)

Let *ptr* be of type *pointer* and $\mathtt{t} \in T_{\mathtt{TABLE},V}$. Then

$$ptr = \iota(\mathtt{t}) \rightarrow searchtree(ptr).$$

Proof by induction on the number of nodes in the list (omitted).

Next we provide the function *lookuptr*:

```
function lookuptr (ky: key; root: pointer): elem;
var   cur: pointer;
      searched: boolean;
begin
      cur:= root;
      searched:= false;
      while not searched do
      begin
            if cur = nil
            then begin lookuptr:= errorelem; searched:= true end
```

```
            else
                if eq(ky,cur↑.k)
                then begin lookuptr:= cur↑.e; searched:= true end
                else
                    if lt(ky,cur↑.k)
                    then cur:= cur↑.l
                    else cur:= cur↑.r
    end
end;
```

Presently, a lemma similar to lemma 4.4.2.b can be formulated. It states that *lookuptr* is well-defined for single additions to a well-formed tree.

4.6.2 Second well-formedness lemma for trees (part b)

Let `k`, `k'` be of sort `KEY`, `e` of sort `ENTRY`, and `t` of sort `TREE`, and let $ptr' = \iota(\texttt{treeadd}(\texttt{k}', \texttt{e}, \texttt{t}))$. Then

$$ptr = \iota(\texttt{t}) \rightarrow$$
$$[eq(\iota(\texttt{k}), \iota(\texttt{k}')) = true \rightarrow lookuptr(\ \iota(\texttt{k}),\ ptr'\ ) = \texttt{e}] \wedge$$
$$[eq(\iota(\texttt{k}), \iota(\texttt{k}')) = false \rightarrow lookuptr(\ \iota(\texttt{k}),\ ptr'\ ) = lookuptr(\ \iota(\texttt{k}),\ ptr)]$$

The *proof* follows from the observation that $ptr' = \iota(\texttt{treeadd}(\texttt{k}', \texttt{e}, \texttt{t}))$ is defined in terms of $ptr = \iota(\texttt{t})$.

According to Theorem 4.5.2.8 it is now sufficient to prove (E the set of equations from module `Tables`):

$$\{\texttt{k} \in \texttt{KEY} \wedge \texttt{tbl} \in \texttt{TABLE} \wedge ky = \iota(\texttt{k}) \wedge root = \iota(\texttt{tbl})\}$$
$$elt := lookuptr(ky, root)$$
$$\{\rho_{\texttt{ELEM}}(elt) \in C_{\texttt{ELEM},O} \wedge \texttt{lookup}(\texttt{k}, \texttt{tbl}) =_{\texttt{E}} \rho_{\texttt{ELEM}}(elt)\}.$$

This follows immediately from lemma 4.6.2, and the definition of $\rho_{\texttt{ELEM}}$.

4.7 Conclusions

The Chapter provides a functionally oriented (black box) approach to the implementation of modular algebraic specifications. The main advantages are listed below.

- It provides a theoretical background for the *separate implementation* of modules.
- The implementation above is based on the *initial* behaviour of certain functions, the observing functions. This provides an intuitively clear semantics.
- A correctness criterion for implementations is given in Hoare logic, allowing the application of *standard optimization techniques.* In algebraic terms this means that functions which are not observing may have semantics closer to those in a *final* model.
- The combination of separate implementation and (hence separate) optimization allows the construction of a *library* of (possibly optimized) modules.

The loss of the *initial algebra semantics* might also be listed as a disadvantage. Terms are only judged different when they have different effects (*confusion* is allowed) and other invisible terms (*junk*) may be introduced. On the one hand, precisely these two "undesirable" effects allow the introduction of optimal implementations. On the other hand, they make the semantics of a module less clear to the user (i.e., someone writing a module importing the optimized module). This problem is minimized by the fact that the criteria for use of the module, allowing the set of observing terms only, are rather easy.

Chapter 5

From Formal Specification towards Derivation: the MacMahon (Swiss) System

The Swiss System is originally developed as a tournament system allowing many participants to play a chess tournament in a limited number of rounds. It avoids both the drawbacks of round-robin tournaments (limited capacity) and knock-out tournaments (early dropouts). The system has been introduced in 1895 by Dr. J. Muller in Zürich. Since that time it has been used in many variations at chess tournaments, and (sometimes adapted to the circumstances) at bridge, draughts and go tournaments as well. The latter variation is called the MacMahon system.

Though many rule sets for the Swiss System try to formulate the proceedings for finding a pairing in an unambiguous way, it proves to be surprisingly hard to find an efficient implementation. This is due to the fact that an algorithm both has to steer clear of computing a combinatorial explosion of possible solutions when there is an abundance of allowed solutions, and on the other hand of missing a solution when there is a scarcity of allowed solutions. In this Chapter an attempt is made to use transformational techniques to find an almost linear algorithm for average cases from a formal specification of the problem.

5.1 Introduction

5.1.1 The Swiss System informally

For many years chess tournaments could be encountered in three forms: the match, the round-robin tournament (each player meets each opponent once), and the knock-out tournament. Obviously the match is limited to two players only, but also the round-robin tournament is not suited to a large number of participants (for $n \geq 2$ and $2n-1$ or $2n$ players one needs $2n-1$ rounds). On the other hand, the knock-out tournament lets most players participate for only one or two games, while the final standing is not very accurate (often a strong player can be eliminated by an unlucky pairing in an early round, while at the same time far weaker participants play each other, one coming through to the next round).

In 1895 Dr. J. Muller introduced a modified knock-out tournament system at a Zürich (Switzerland) chess tournament [Kaž80], which has evolved into what is known today as the Swiss System. The basic idea is as follows:

1. in every round, each player is paired with an opponent with equal score (or as nearly equal as possible);

2. two participants are paired at most once;

3. after a predetermined number of rounds the leading player wins.

So for round one either a random pairing is made, or some seeding system is used. In round two all winners play each other. The same holds for the players with a draw, and for the losers. If there is an odd number of winners one of them plays a person with a draw (or a loser, but only when there are no people with drawn games). In round 3 players with 2 points play each other, players with $1\frac{1}{2}$ points, etc. Again, if we have an odd number of players then one is selected to play someone of an adjoining group.

This system has the advantage of a knock-out system in the limited number of rounds. The disadvantages mentioned above have disappeared. Nobody is eliminated, and a chance to come back after an early loss exists. On the other hand, final standings in the middle groups are still not very accurate when compared to a round-robin tournament. This cannot be fully remedied by adding extra rounds to the tournament, since this mainly adds to the accuracy at the ends of the group, the places where the tournament most resembles a round-robin between the top, resp. the bottom, players.

Many variations exist, mainly to accomodate for particular circumstances. For instance, in chess tournaments players heavily favour playing with the white pieces. Hence colour allocation should be treated fairly to ensure fair competition. Or participants may not wish to play their own clubmates since they can do so at home. More tricky restrictions exist in top chess tournaments, where players may wish to compete for the title of Master or Grandmaster, for which they need to play a certain number of strong opponents during the tournament.

5.1.2 A difficult problem

Various attempts have been made to write an algorithm for the Swiss System. However, a really satisfactory solution is, to our knowledge, still non-existent. Van den Herik [Her88] reported on a not completely satisfactory attempt by some students, called ZORBA. Here even the formulation of the problem has been influenced by the programmers, which resulted in a new version of the Swiss System [GH88].

Similar problems have been studied in combinatorics (see textbooks as [PTW83]), e.g., the problem of allocating people to jobs in the most efficient way. Unfortunately, for our problem, which can be seen as a generalization of the job allocation problem, there is no known efficient solution. For n players the best solutions are $\mathcal{O}(n^3)$. Variants of this solution have been implemented by Jansen and Kindervater [Her91] and Gerlach [Ger94]. They claim reasonable running times in practical cases, 60 players for the former and 150 (with a claim for 300) for the latter, even though Gerlach's variant is $\mathcal{O}(n^5)$. The algorithm presented here has run a tournament with about 500 players.

5.1.3 An overview of the Chapter

Some notes to the informal description from Chapter 2 are given in section 5.2. A sketch of the corresponding formal description follows in section 5.3. The central part of the paper is the discussion on the introduction of domain knowledge and the consequences for the specification and implementation in section 5.4. Some variations of the Swiss system together with their implications for specification and program are given in section 5.5. This is followed by a brief description of the actual implementation (section 5.6) and the conclusions. Excerpts from the source code can be found in an annexe.

5.2 The informal description

The informal description of the Swiss system can be found in Chapter 2 (section 2.2.1). As a reminder to the reader, the Swiss System has been selected there because a real-world problem helps in keeping a fairly unbiased view of the subject. Furthermore, a good informal specification of this problem is known, which is not too lengthy, and sufficiently complete for the purpose of human application. Also, it is not one of the "insider problems" of computing science, which would cloud the discussion with various standard solutions.

Comments on the specification

It should be noted that the description in Chapter 2 is a fairly complete algorithm for human application. As such it has various redundancies. A good example is rule 6, which is essentially a special case (only one score group, no illegal pairings possible) of rule 9.

Also, the description is incomplete. A clear case is described in the note at the end of rule 9b. But surprisingly enough, even the Basic Laws in rule 5 are ambiguous. Suppose at a certain stage of a hypothetical (but quite plausible) tournament the top score group has three players, A through C, and the next group two players, D and E.

ranking on top

rank	players	score
1.	A, B, C	k
4.	D, E	$k-1$
6.	F, etc.	$k-2$

Now (the 1st option) C is selected as odd man and A plays B; C has already played D and E, so they play each other, and C plays someone, say F, two groups below. However, if B could play D – or symmetrically E – (2nd option), B could have been selected as odd man. Then A plays C, B plays D, and E plays someone from the group containing F.

possible pairings

	1st option			*2nd option*	
game	difference	same score	game	difference	same score
A–B	0	2	A–C	0	2
D–E	0	2	B–D	1	0
C–F	2	0	E–F	1	0
total	2	4	total	2	2

The original intention of the Swiss System is that the latter pairing is preferable, but the former has more players playing someone with the same score (rule 5b), while the pairings of players B first and C in the second alternative are the best possible according to rule 5c.

5.3 The formal specification

For the formal algebraic specification we assume some basic data types with their functions available and specified in the usual way. These include the `Booleans`, `Naturals`, `Integers`, `Characters`, and `Strings` (of `CHAR`), with sorts `BOOL`, `NAT`, `INT`, `CHAR`, and `STRING`, respectively. We will use the algebraic specification formalism ASF [BHK89] with some syntactic sugaring for common operators.

In the following section the basic framework for the description of a valid pairing will be made. This includes the specification of a set of `PLAYER`s, and possible `GAME`s. Many ways to model this structure are available (see Chapter 2). Our choice corresponds to the usual description in combinatorics [PS82, PTW83], which uses weighted graphs. In those terms a player is a *node* and a possible game is a weighted *edge*.

The weight of a game is an indication of its non-desirability. A weight of 0 indicating a perfect match, any positive number is less desirable. Excluded games (games between players who have met in a previous round) have infinite positive weight.

Weights are ≥ 0, since the problem is commonly formalized in combinatorics as an optimization based on reducing the sum of the weights on a number of edges in the graph. Negative weights could be allowed only if edges were not counted twice. This problem is elegantly avoided by including positive weights only, since then double-counting cannot lower the sum.

To allow for different approaches with pairing systems the weights are made explicit as functions from `GAME` to some value in `NAT` rather than being an implicit attribute of the datatype `Game`. Hence presentation of the full weighting can be left off to the moment when a formal description of a proper pairing is given.

5.3.1 The basic datatypes for a tournament

The following basic datatype declarations are one way of describing a tournament. For other choices see Chapter 2. It is argued therein that whatever method is chosen it should in one way or another include players and relations (games) between those players. Hence at the description level this is what we need and no more. To describe the module `Tournament` we need its sub-modules `Players` and `Games` below.

```
module Players
begin
  exports
    begin
      sorts PLAYER
      functions
        first-player  :                  -> PLAYER
        next          : PLAYER           -> PLAYER
        no-participant:                  -> PLAYER
        player-number : PLAYER           -> NAT
```

```
        rating          : PLAYER          -> NAT
        name            : PLAYER          -> STRING
        eq              : PLAYER # PLAYER -> STRING
    end

  imports Boolean, Natural, String

variables
  p, p1, p2: -> PLAYER

equations:
[p1] player-number(first-player) = 1
[p2] player-number(next(p))      = succ(player-number(p))
[p3] eq(p1,p2) = eq(player-number(p1), player-number(p2))

-- the following equations depend on actual players,
-- for example:
[p4] name(first-player)          = "Gary Short"
[p5] rating(first-player)        = 2345
[p6] name(next(first-player))    = ....

end Players
```

For convenience players are enumerated via functions **first-player** and **next**, with end marker **no-participant**. This enumeration is not implicitly coupled to the player number, though (again for convenience) it is specified that way. Player numbers could as well be distributed randomly, as long as they are unique. The functions **rating** and **name** are of course dependent on the actual players involved. An example is added to the specification.

Next to be specified is straightforward module **Games**.

```
module Games
begin
  exports
    begin
      sorts GAME
      functions
        game          : PLAYER # PLAYER -> GAME
        is-in         : PLAYER # GAME   -> BOOL
        first-player  : GAME            -> PLAYER
        second-player : GAME            -> PLAYER
    end
  imports Boolean, Natural, Players

variables:
  p, p1, p2: -> PLAYER
```

```
  g        : -> GAME

equations:
[g1] first-player(game(p1,p2))  = p1
[g2] second-player(game(p1,p2)) = p2
[g3] is-in(p, g)                = eq(p, first-player(g)) OR
                                  eq(p, second-player(g))

end Games
```

It is a bit inconvenient in this specification that no direct provision can be made within initial algebra semantics to state that the players in the game must be two different players, because **game(p,p)** remains a correct term. It could be excluded through the introduction of a special predicate **legal-game**. Instead, such games will be excluded in the specification of predicate **legal-pairing** in the specification of module Round.

```
module Tournament
begin
  exports
    begin
      sorts PLAYER, GAME
      functions
        score          : PLAYER          -> NAT
        weight         : GAME            -> NAT
        scoremismatch  : PLAYER # PLAYER -> NAT
        wtscore        :                 -> NAT
        systemmismatch: PLAYER # PLAYER -> NAT
        equalmismatch  : PLAYER # PLAYER -> NAT
        oddmanmismatch: PLAYER # PLAYER -> NAT
        wtsystem       :                 -> NAT
        colormismatch  : PLAYER # PLAYER -> NAT
        wtcolor        :                 -> NAT
        color-pref     : PLAYER          -> INT
        localmismatch  : PLAYER # PLAYER -> NAT
        wtlocal        :                 -> NAT
        halfgroup      : PLAYER          -> NAT
        rankingroup    : PLAYER          -> NAT
    end

  imports Boolean, Natural, Integer, Games, Players

variables
  g      : -> GAME
  p1, p2: -> PLAYER
  n1, n2: -> NAT
```

```
  i1, i2: -> INT

equations

[t1] weight(g) = if(already-played(p1,p2),
                    infinity,
                    wtscore  * scoremismatch(p1,p2)  +
                    wtsystem * systemmismatch(p1,p2) +
                    wtcolor  * colormismatch(p1,p2)  +
                    wtlocal  * localmismatch(p1,p2) )
      where p1 = first-player(g),
            p2 = second-player(g)

[t2] scoremismatch(p1,p2) = square(diff(score(p1),score(p2)))

[t3] systemmismatch(p1,p2) = if(score(p1) = score(p2),
                                equalmismatch(p1,p2),
                                oddmanmismatch(p1,p2))

[t4] equalmismatch(p1,p2) = abs(halfgroup(p1) -
                        abs(rankingroup(p1)-(rankingroup(p2)))

[t5] oddmanmismatch(p1,p2) = if(score(p1) > score(p2),
                        2*halfgroup(p1) - rankingroup(p1) +
                        rankingroup(p2) - 1,
                        oddmanmismatch(p2,p1) )

[t6] colormismatch(p1,p2) = if(i1 = 0 OR i2 = 0 OR
                               NOT (sign(i1) = sign(i2)),
                               0,
                               abs(i1) + abs(i2))
       where i1 = color-pref(p1),
             i2 = color-pref(p2)

end Tournament
```

For the sake of brevity of presentation trivial functions like **wtsystem** (a constant) and **color-pref** have not been specified above. Most of them may be easily specified in detail or even (the weights) be left to the end-user. Note that negative weights do not exist, so we can use **NAT**urals.

Looking at equation **t1**, test **already-played**, explained in more detail in section 5.3.2 below, corresponds to rule 5a, while rules 5b and 5c are coded in the function **scoremismatch**. Note the square in equation **t2** to avoid the problem sketched at the end of the last section. Also, the score is represented as a natural number, even though chess tournaments include half points for draws. This is usually not the case in go tournaments where only full points can be won. However, also in chess

tournaments the set of possible score is a subset of the rationals ($\{0, 0.5, 1, 1.5, ...\}$) which is isomorphic to the integers under addition. So representing those with double their value is correct, if somewhat opaque.

`systemmismatch` represents the specification of a combination of rules 6 and 9b (in `equalmismatch`) and rules 9a (with 10-12) in `oddmanmismatch`. Rules 13-17 will be represented in `colormismatch`. These functions use a variety of simple counting functions: `score`, `rankingroup`, `halfgroup` (half of the number of players in the same score group), and `colour-pref`(erence), which depend on results per round. Hence they are not specified here, analogous to the player names and ratings in module `Players`.

Two functions need some additional commentary (see also Chapter 2):

- Weight `wtscore` is much more important than all other weights combined. The best way to specify this is to use a tuple for the weight, ordered lexicographically, where `wtscore` is the dominant factor. However, this also means that connections to the current mathematical theory on optimization ([PTW83], [PS82]), which is single-valued for easy addition of weights, are lost.

  This can be retained when maximal values on the contributions of the other mismatch weights exist, possibly by proof from the equations, but if need be by postulation. Then `wtscore` can be chosen greater than the combination of these maximal values to obtain the same ordering.

- The function `localmismatch` is not specified here, because too many possibilities for contributing factors exist. The role of this function is to allow for pairing biases peculiar to a few specific tournaments only. An example is the question whether players from the same club are paired against each other if it can be avoided. Some people do not like it, but for the fairness of the tournament it might be better, so others prefer to take such considerations into account.

5.3.2 The pairing specification

Next we have to specify what is a correct round. Actually, the last step, from one round to a number of rounds, is abstracted in `already-played`, which could be seen as parameterized also with a list of previous rounds. In this way we can focus on the real problem of the Swiss system: finding the pairing for the next round, while abstracting from administrative details.

```
module Round
begin
  exports
    begin
      sorts ROUND
      functions
        emptyround  :                  -> ROUND
        addgame     : GAME # ROUND     -> ROUND
```

```
        nextgame     : ROUND          -> GAME # ROUND
        isinround    : PLAYER # ROUND -> BOOL
        total-weight: ROUND           -> NAT
        full-pairing: ROUND           -> BOOL
        best-pairing: ROUND           -> BOOL
    end

  imports Boolean, Natural, Tournament

variables rnd, rnd1, rnd2 : -> ROUND
          gme, g1, g2     : -> GAME
          player, p1, p2  : -> PLAYER

equations

[r1] total-weight(rnd) = if(eq(emptyround,rnd), 0,
                            weight(gme) + total-weight(rnd1) )
          where (gme,rnd1) = nextgame(rnd)

[r2] isinround(player,emptyround) = false

[r3] isinround(player,rnd) = player = first-player(gme) OR
                             player = second-player(gme) OR
                             isinround(player,rnd1)
          where (gme,rnd1) = nextgame(rnd)

[r4] nextgame(emptyround) = (nilgame, emptyround)

[r5] nextgame(addgame(gme,rnd)) = (gme, rnd)

[r6] full-pairing(rnd) = legalpairing(rnd) AND
                         if(NOT isinround(p1,rnd) AND
                            NOT isinround(p2,rnd),
                            p1 = p2,
                            true)

[r7] legalpairing(emptyround) = true

[r8] legalpairing(rnd) =
             NOT (first-player(gme) = second-player(gme)) AND
             NOT already-played(first-player(gme),
                                second-player(gme)) AND
             NOT isinround(first-player(gme),rnd1) AND
             NOT isinround(second-player(gme),rnd1)
      where (gme,rnd1) = nextgame(rnd)
```

```
[r9] best-pairing(rnd) = full-pairing(rnd) AND
             leq(total-weight(rnd), total-weight(rnd2))
      where full-pairing(rnd2) = true

end Round
```

For a correct round, everybody has to be paired (**fullpairing**). Then, if a pairing exists, we need the best one, according to the weights. This results in an 'open' equation for **best-pairing**, which may be seen as universally quantified over all possible other rounds **rnd2**. This is shorthand for a complete enumeration of all possible pairings, so the number of closed equations in this specification when fully written out would be still finite, even though the specification would become very much longer.

Relation between the specifications

In summary, the structure of the relation between the formal and the informal specification is as follows:

- basic datatypes **PLAYER** and **GAME** are introduced, together with their relevant attributes;
- next, the pairing system is formalized in the **weight** function and its auxiliary functions and constants;
- finally, a correct **ROUND** can be stated as a simple minimization problem for each round.

This is not the only way to structure the specification. **ROUND** is also a prime candidate for a basic datatype. The reason for the current choice is based on declaration-by-need, i.e. rounds are only needed when you want to describe a pairing.

5.4 Derivations at the specification level

A trivial implementation based on equation `r9` can be made as follows. Generate all possible sets of pairings, filter those until all pairings are legal, and minimize on the sum of the weights. This is indeed possible in theory, and even practical for small numbers of players. However, due to the multitude of potential possibilities for pairings this will not lead to an efficient solution.

One possible route to a satisfactory implementation is of course stepwise transformation of the program resulting from the process above. But it is much more economical to transform the original specification, since programs tend to be *longer*, and programs tend to be *less readable*. Both these reasons occur because of the need to introduce fairly low-level implementation details.

This section deals with some ways to gear the original specification towards a more efficient implementation. This is done by taking advantage of *domain knowledge*, especially knowledge about score groups as they appear in the Swiss system.

5.4.1 The need for specialization

Initial complexity analysis of the pairing problem shows that for $2n$ or $2n-1$ participants ($n \geq 2$) the number of possible pairings is: $\prod_{k=1}^{n}(2n-2k+1) \geq \prod_{k=1}^{n}(n-k+1) = n!$ So trying all possibilities leads to a non-polynomial algorithm, which is impractical.

The generalized problem is known in combinatorics as a *weighted matching problem*, see for instance the text books [PTW83], [PS82]. For a subclass of those problems, the so-called bipartite matching problem, reasonably efficient ($\mathcal{O}(e\sqrt{n})$ or $\mathcal{O}(n^3)$ for n the number of nodes and e the number of edges) solutions exist. In this subclass one has two groups (hence the name *bipartite*) and pairs have to be made with one member from each group. But the Swiss pairing problem does not allow for such an easy division into two groups, so this does not offer a way out.

Instead, we can try to directly minimize the sum of the `cost` function results for individual games. Obviously, this involves elimination of all illegal pairings. Further progress cannot be made, however, without detailed knowledge about the structure of the `cost` function, i.e. knowledge about the most significant contributing factors, which then can be reduced effectively.

The only viable alternative is sorting the possible games for each player according to weight. This pre-processing is rather costly, not only using $\mathcal{O}(n^2)$ space for n players, but also needing n sortings of length n, costing $\mathcal{O}(n^2 \log n)$ time. It is conceivable that with a good choice of sorting algorithm this can be improved to $\mathcal{O}(n^2)$ time on average, but this choice will need domain knowledge also. This direction has not been investigated further, since the solution presented below does not need the extra space, and is already close to the better $\mathcal{O}(n)$ in time on average.

In general the introduction of domain knowledge does detract from the value of the algorithm in the sense that it is less general, and hence less commonly applicable. This cannot be denied, but on the other hand the general problem (weighted matching) is only solved in order $\mathcal{O}(n^3)$, not sufficiently efficient for this purpose.

So either this general problem must be solved (and it is likely it cannot be improved upon), or we have to resort to using what domain knowledge is available, hence turning to a more specialized problem.

A more serious problem is the fact that the solution presented might stop with a sub-optimal pairing. This is discussed below.

5.4.2 Domain knowledge

It remains to select a good choice for the proper domain knowledge to be used. We have two pointers to the same heuristic. Both the human algorithm (i.e., the way a tournament director operates when he runs a tournament by hand) and the informal specification indicate that *score group* is the main initial selection criterion.

This is a promising heuristic. Score groups may be small, so the number of possible combinations is not very large and we can try them all. Or score groups can be large, but then only relatively few pairings will be disallowed, so directed search for optimal pairings is possible.

An as yet unexplored area in between exists when score groups are about the size of the number of rounds played before. Then it is possible that quite a few mutual games might already have been played before, which will complicate the pairing due to the reduced number of correct possiblities. However, firstly for practical purposes the number of rounds is usually below 10, so such groups number not much above 10 players. And secondly, every disallowed pairing reduces the number of combinations possible. This reduction is initially quite large: for $2n$ or $2n-1$ players the first illegal pairing reduces the total number of possible pairings by $\frac{1}{2n-1}$ times this total. So if need be it becomes feasible to consider all legal pairings in the calculation.

How are score groups used

Introduction of score groups in the specification is rather straightforward. Within each score group, starting from the top, an optimal pairing is found. Possible odd men are taken along to the next group. If the pairing attempt for the last group fails it is combined with the previous group(s) until a legal pairing can be found.

For reasons of speed this scheme is modified by the size of the group:

- for *small groups* (less than or equal to l_{full} players) all possibilities are exhaustively tried;
- for *intermediate groups* (above l_{full} and les than or equal to l_{fast} players) all possibilities to pair the top half versus the bottom half are tried – if the optimal pairing in this way is a legal pairing (i.e., all games in the pairing are allowed), that one is chosen, otherwise the exhaustive algorithm is tried;
- whithin *large groups* (more than l_{fast} players) the algorithm looks for the most expensive game (usually an illegal one) and tries to improve upon that cost level by using the best exchange of players in the bottom half – again, if the optimal pairing in this way is legal, it is chosen, otherwise the algorithm for intermediate groups is tried.

These border values l_{full} and l_{fast} were initially experimental. The choice of 6 for l_{full}, resp. 10 for l_{fast}, proved satisfactory.

When groups are combined from below upwards, not many legal pairings exist. Hence an exhaustive backtrack search with the legality of all games as a break-off criterion is feasible on the combined group.

Sacrificing generality

The main problem with the division of the complete group of players is the fact that there is no way to guarantee that the sum of the optimizations of the parts is also optimal for the whole. Again, the only way to be sure is to calculate one's way through all possibilities, which is obviously too expensive. So while optimal pairings in the general sense cannot be guaranteed, within one score group they can be found. For validation (the players' general perceptive of the correctness) this is what matters.

The specification can be adapted as follows: games are grouped per score group (choose the lower score in case of different scores within a game), and within these score groups a separate minimization is conducted. This minimization is then lexicographically ordered from the highest score to the lowest. An additional constraint on combined groups is that the number of oddmen is minimal. Other functions remain unchanged.

```
module Round
...
  exports
...
    best-pairing          : ROUND # NAT -> BOOL
    weight-per-scoregroup: NAT # NAT   -> NAT
    minimum-nr-oddmen     : ROUND # NAT -> BOOL
    highestscore          : ROUND       -> NAT
...
variables rnd, rnd1, rnd2 : -> ROUND
          gme, g1, g2     : -> GAME
          player, p1, p2  : -> PLAYER
          scre            : -> NAT

equations:
[r1] weight-per-scoregroup(scre,rnd) =
                         if(eq(emptyround,rnd), 0,
                            if(eq(scre,min(score(firstplayer(gme),
                                           score(second-player(gme))),
                               weight(gme),0)
                            + weight-per-scoregroup(scre,rnd1))
...
[r7] best-pairing(rnd) = full-pairing(rnd) AND
                         if(full-pairing(rnd2),
```

```
                        best-pairing(rnd,rnd2,highestscore(rnd)),
                        true)

[r8] best-pairing(rnd,rnd2,scre) =
                      if(minimum-nr-oddmen(rnd,scre) AND
                        minimum-nr-oddmen(rnd2,scre),
                        leq(weight-per-scoregroup(scre,rnd),
                            weight-per-scoregroup(scre,rnd2)),
                        true)) AND
                      if(scre>0, best-pairing(rnd,rnd2,prev(scre),
                        true)

[r9] minimum-nr-oddmen...
[r10] highestscore...
end Round
```

This specification contains the same universally quantified variable **rnd2**. Again, this variable can be eliminated at the cost of a complete enumeration.

5.5 Some possible variations in the pairing system

Over the years, many variations of the Swiss System have been tried in practice. Some more important practical variations and their effect on the specification and implementation are discussed below.

The ease of implementation depends on the degree of deviation from a true Swiss system. But existing variations are designed for easy application by hand, so major difficulties are not to be expected.

5.5.1 Random draw

The pairing is of course at the heart of any Swiss System. To obtain a *manual* pairing a random draw within a score group is undoubtedly the fastest. Random draws are also used to select odd men. This is easily specified by changing functions `equalmismatch` and `oddmanmismatch` to constant value 0. Those functions are the places where any ordering within scoregroups is relevant.

Of course, implementation is rather easy, since only score difference should have a positive weight. However, one *caveat* is in order. 'Random' is not the same as 'anything is ok'. The trivial first attempt, pairing 1 vs. 2, 3 vs. 4, etc., is a system in its own right. And it is especially detrimental to the course of the tournament if player numbers are given out sorted on rating. Effectively the projected 'final' between nrs. 1 and 2 is then automatically scheduled for the first round.

Pseudo-random variations of the Swiss system also exist. They mainly try to avoid the pitfall mentioned above. At the same time the efforts of the tournament director are more mechanical, and hence less prone to allegations of prejudice, which sometimes happens with a truly random draw. In practice however this way of pairing is meant for human application when time between rounds is very limited. A computer should be able to do better than such a crude algorithm.

In general it is easy to adapt the specification for other variations. The implementation in Annexe A is based on another common variation [GH88]:

```
[t4] equalmismatch(p1,p2) = abs(rankingroup(p1) +
              rankingroup(p2) - 2 * halfgroup(p1) )
```

Here player 1 is preferably paired with player n, player 2 with player $n-1$, etc., within the same scoregroup.

5.5.2 Special circumstances

Sometimes the Swiss system is adapted to accomodate for certain wishes of the players. This is often a very subjective matter, so it is difficult to give general guidelines for implementation. Two typical cases are given below.

- A bias of the draw towards avoiding certain clashes, e.g., between members of the same club (who play each other often enough), or towards forcing certain meetings (e.g. for publicity reasons). This should be avoided in the top of the tournament, but it is often preferred by players who are in the tournament just for entertaining themselves.

  Implementation has been done by introduction of weightings dependent on the desirability of the game w.r.t. these considerations and the score of the players involved.

- In top-level chess Swiss tournaments many players aim for so-called title results. They are needed for obtaining various master levels, like International Master and International Grand-Master. Basically a player has to perform well in a number of tournaments to be awarded a title. How well depends on the title involved.

  Such a title result is dependent on various factors, of which the score is of course most relevant, but what score is needed depends on the opposition. Also, certain minimum levels in the number of titled opponents and foreign opponents have to be met. This does become very complicated.

  Moreover, it depends on the possible pairings in future rounds, so it is a bit beyond the scope of the current algorithm, which only looks at the past. A solution might be to establish minimum levels of opposition per player per round derived from the end levels needed. But this is essentially still an open problem.

5.5.3 Go tournaments: the MacMahon system

A special variation of the Swiss System is used in Go tournaments. It is treated here because the practice runs to validate the weightings of the algorithm have been made on go tournaments using this variation.

Every amateur go player is graded according to his playing strength in approximately equal steps. A similar, but unrelated, rating exists for professional players. A top amateur player is graded 6-dan, then it goes down to 5-dan, ..., 1-dan, 1-kyu, 2-kyu, ... and so on to about 20-kyu. These grades are used to give each player a start score prior to round 1. If, say, all 1-kyu players start at score s, then all 2-kyu players will start at $s-1$, and all 1-dan players at $s+1$. This is usually adjusted at the top to allow for a decent group of players at the same top level, by giving them the start score for the lowest among them. Often the same type of adjustment is implemented at the bottom.

Then a normal 'Swiss' tournament is run, with the proviso that, except at the top, prizes are given for the number of wins rather than for the final standing. This is needed since the start scores are usually further apart than the maximum number of rounds so someone at the bottom cannot win the tournament, but (s)he should be able to win a prize.

The name of this variation, MacMahon system, comes from a certain Lee MacMahon (or McMahon), who has introduced a similar system around 1960. But precise details have been lost in time.

Of course, the MacMahon system can be adapted to chess and other rated events quite easily, e.g., 100 Elo-rating points in chess is 1 starting point extra. However, this has not yet been done to my knowledge.

Conceptually this is an accelerated variation of the Swiss system, a known but rather unusual practice in chess. Accelerated pairing systems make use of the fact that often the first round is almost a foregone conclusion when players are ranked on playing strength to avoid clashes in the top. A typical variation is described in [Kaž80]: the rank-point system of Haley. In this (rating-based) system the top half of the players get 1 point extra for the pairing of the first two rounds (only). So by temporarily changing the score of the top players they play each other immediately, thus speeding up the tournament.

5.6 Implementing the specification

With the specification in the modified filter format described in Section 5.4.2 it remained to generate relevant pairings for the standard generate-and-test implementation. The filter here is a legality check of the pairing per score group followed by minimization.

The top level algorithm has the following simple structure:

```
WHILE still a score group remaining
DO find an optimal legal pairing for the highest remaining
     group;
   IF a legal pairing has been found
   THEN handle any odd men
   ELSE add a neighbouring group to this highest group
   FI
OD
```

The search for an optimal and legal pairing itself is implemented, depending on the size of the group, as one of three generate-and-filter algorithms, of successive levels of thoroughness as described in section 5.4.2.1. Each algorithm is analyzed on speed in section 5.6.3.

The neighbouring group is usually the next lower score group. However, if the lowest score group cannot be paired legally, the next higher group is added.

5.6.1 The case study: MacMahon tournaments

For algorithms like the Swiss system (or rather, like the pairing algorithm within the Swiss system) real and perceived correctness do not necessarily overlap. A mathematically correct pairing may be lacking from the human perspective. The problem lies with construction of the specification: has the right result been specified? If it is, then a proof can be given, possibly by construction as sketched above.

But there is no way to be sure about the *weights* in the specification without a proper *validation* of those weights. Their initial choice may very well turn out to be wrong. So a practical application of the implementation was needed.

An opportunity for the validation has been found in the world of go, where especially the MacMahon variation (see section 5.5) has gained widespread application. The algorithm has been fine-tuned for the particularities of go (like the reduced relevance of colour allocation) by controlling the weights in the specification. This algorithm has been built into a program developed by students at the University of Nijmegen, GoMMTour [GT93], an organization support program for go tournaments. The experience at about twenty tournaments (appropriately enough among them one in Zürich!) has shown that the pairings generated are of good quality.

The main advantage of the use of MacMahon tournaments for validation is the relatively large dependence on the small group algorithms, since score groups are easily small enough to defy finding a pairing within the group. Hence for validation purposes this has been a valuable experience.

A disadvantage of go tournaments is that interesting effects which are important in chess have not been tested. Especially colour allocation could pose a potential problem. The game of go is too well balanced to make this a major cause of dissatisfaction. Also, draws are either eliminated or rare in tournament go, and this reduces the number of players in the same score group who have already met. The influence of this on the behaviour of the algorithm is unclear.

A final word on chess: the current algorithm provides no easy way towards implementation of title result directed pairings, except of course for pairing the last round. It might be possible to find a specification for `localmismatch` allowing for this, but too many factors influence a player's chances when possible pairings in later rounds than just the current round are involved. Fortunately, allowing for title results happens in a few tournaments only. Hence it is not too much of a loss to have to revert back to manual pairing in those cases.

5.6.2 Correctness and weight validation

During several practice runs on go tournaments in The Netherlands, Germany and Switzerland in the period between the Fall of 1992 and the Spring of 1993 some mistakes in the pairings as perceived by the users were encountered. These all had to do with the weights for the handling of odd men, which were estimated too low relative to the weight for colour allocation. Since perception of correctness plays an important role, validation of the results was necessary to find such problems. The algorithm itself is correct by construction, but the specification was apparently too complex to detect all interactions.

The reason for this introduction problem is that the relative values are rather loosely described in the informal specification. But the corrections were elegantly implemented by changes to the relevant weight constants in the formal specification, and the corresponding changes of these constants in the implementation. Hence the algorithm remained essentially unchanged.

It was also possible to make the adjustment of these weights available to the user, but, due to the small size of the target computer (see 'further constraints' in section 2.1) the cost function had to be curtailed more strictly than expected. The value of only 1000 for games already played (the value `infinity` in the specification) proved rather too low for small tournaments, where the quadratic development of the score-difference function resulted in larger numbers for legal games. Usage of unrestrictedly large naturals would have solved that, but only at a (probably unacceptable) reduction in speed.

5.6.3 Speed

Speed proved to be satisfactory. No measures have been taken, but the combination of fast algorithms for large groups and complete algorithms for small groups resulted in fast pairings. This can be argued as follows:

- For groups on or below a certain small limit l_{full} all possible pairings are calculated through, but this results in a fixed maximum effort of $(l_{full} - 1) \times$

$(l_{full} - 3) \times \cdots \times 1$ possible pairings to be examined, which is a constant since l_{full} is constant. (Our experimental first choice ($l_{full} = 6$) proved to be satisfactory.)

- For groups above l_{full} but on or below limit l_{fast} only the bottom half of the pairings are interchanged so the maximum number of pairings possible is $\left\lfloor \frac{1}{2}(l_{fast} + 1) \right\rfloor !$, which is again a constant. (The experimental first choice ($l_{fast} = 10$) proved to be satisfactory.)

- So it remains to investigate the behaviour for groups with more than l_{fast} number of players. The algorithm tries to find the optimal improving exchange for the most expensive (i.e., least desirable) game. Since no more exchanges will be made than the total weight for this score group (every exchange must be improving) this way of reducing the weight proved to be fast enough.

Even on personal computers pairings for tournaments up to 200 players were completed in a matter of seconds. One exception has been found to this rule: backing up over sparse groups in the lower part of a tournament in later rounds can be rather time-consuming. Then groups can be formed which are substantially greater than l_{full} in size. In this case however, unlike the situation above, all possible pairings will be tried because the groups are combined. On the whole this was not too expensive, due to the fact that combination only occurs when few legal combinations are available to begin with.

5.7 Conclusions

The experiment with the Swiss System shows the following:

- Even though a description exists sufficient for human application it proved to be very hard to get a complete specification of the pairing in the Swiss system. Indeed, problems which also have to take into consideration pairings in subsequent rounds have not been solved in the current framework.

- Modifications (in this case a weakening) on the specification level proved to be satisfactory, both because they allow for easy validation along the usual route if the specification would change substantially, and because high level constructs are relatively easy to handle.

- The introduction of domain knowledge resulted in an efficient algorithm, with $\mathcal{O}(n)$ efficiency on average, in spite of the fact that the general case is only solved in $\mathcal{O}(n^3)$ (for details see [PS82]).

- Different approaches depending on the size of the problem duplicated effort since more algorithms had to be implemented for originally the same problem. But this also allowed for the efficient use of domain knowledge.

- The modular approach allowed us to focus on the more difficult problems because they could be isolated in specific equations. Hence directed optimization, aimed at handling those problems, was possible.

- Another advantage of the modular approach is the ease with which one specification of the optimal pairing can be replaced by another. The derivations towards programs then also allowed repetition of choice of implementation method due to the similarities in the specification.

From this we can learn and confirm the following lessons for program development in general:

- Formal problem description is still a difficult task. In [DP91] it has been argued that automatic support is desirable, for instance for the detection of overspecification. Since without proper knowledge of the product needed the chance on developing the right program is virtually nil, writing correct specifications is essential. Hence search for specification support should have top priority in Software Engineering research.

- It pays to make derivations, even weakenings, already at the specification level. This is cost-effective for two reasons. Almost always specifications are shorter than the resulting programs, hence modification is less work there. And the line back to the customer for approval or validation is as short as possible.

- While the solution of general problems is mathematically more satisfying and elegant, domain knowledge may result in spectacular improvements in behaviour. Hence it may be worth the extra work to turn this knowledge to the advantage of the programmer.
- The introduction of specialized algorithms for small or border cases (often those coincide) is a worthwile method for speeding up programs.
- The score group approach has some similarity to partial evaluation. In partial evaluation one known, probably dominant, value is used to produce a more specialized, and hence potentially more efficient version of the algorithm. In the approach presented the dominant value of the cost function is factored out by forming groups where the value is the same. This makes it possible to focus on the application of the less dominant factors on a much smaller set of values.
- A final optimization attempt has not been made. Some smart sorting of the relative cost values of potential games, or of the players, may result in a more efficient algorithm for specific types of tournaments. In chess tournaments colour allocation is much more important, so sorting on black/white preference is a good candidate for a directed optimization.

Annexe: Selected parts of the pairing algorithm

This algorithm forms part of a larger Pascal program. Hence some functions, especially those accessing player data, are not shown here.

```
{ Assumptions:
  - if NrOfPlayersToBePaired is odd, add "dummy"
  - dummy is a non-existing CONTESTANT
  - players is an array of CONTESTANTs (Players) to be paired, sorted on
    MacMahon score, SOS/SODOS, and finally rating                        }

{ Local procedures, among others:}

FUNCTION Cost (a, b : CONTESTANT): INTEGER;
BEGIN
   IF (a = Dummy) AND FreeBefore(b) THEN Cost := MaxCost
   {this depends on lazy left to right evaluation of operator AND}
   ELSE
   IF (b = Dummy) AND FreeBefore(a) THEN Cost := MaxCost
   {this depends on lazy left to right evaluation of operator AND}
   ELSE
      IF AlreadyPlayed(a,b) THEN Cost := MaxCost
      ELSE Cost := WtScore  * ScoreMismatch(a,b)  +
                   WtSystem * SystemMismatch(a,b) +
                   WtColor  * ColorMismatch(a,b)  +
                   WtLocal  * LocalMismatch(a,b)
END; {Cost}

{ Backtrack-procedures:}

PROCEDURE FirstPairingAttempt(t,b,topgame,botgame : INTEGER);
VAR i,j: INTEGER;
BEGIN
   { Variation # 1: top plays bottom, top+1 plays bottom-1, etc.
     Used for pairing on SOS/SODOS or on ranking }
   i := t;
   FOR j := topgame TO botgame
   DO BEGIN
      pairing[j,1]:=players[i]; i:=i+1
   END; {OD}
   FOR j := botgame DOWNTO topgame + 1
   DO BEGIN
      pairing[j,2]:=players[i]; i:=i+1
   END; {OD}
   IF b MOD 2 = 0
   THEN pairing[topgame,2]:=players[i] {no odd-man}
   ELSE pairing[topgame,2]:=Dummy {pairing[topgame,1] is first
                                   try for odd-man}
```

```
.....
END; {FirstPairingAttempt}

FUNCTION LegalPairing (topgame,botgame: INTEGER): BOOLEAN;
{ A legal pairing is any pairing wherein no players who have already
  met in previous rounds (Cost will be MaxCost then) meet again. }
....
END; {LegalPairing}

FUNCTION FastSearch (t,b: INTEGER): BOOLEAN;
{ FastSearch only tries improving exchanges between players in the lower
  half of the score group, as they can be found after application of
  procedure FirstPairingAttempt in array Pairing[topgame..botgame, 2]. }

VAR topgame,botgame,i,j: INTEGER;
BEGIN
   topgame := (t+1) DIV 2;
   botgame := (b+1) DIV 2;
   FirstPairingAttempt(t,b,topgame,botgame);
   REPEAT
      i:=FindMostExpensiveGame(topgame,botgame);
      j:=FindBestExchange(i,topgame,botgame);
      IF j>0 {j=0 means no improving exchange possible}
      THEN Exchange(pairing[i,2],pairing[j,2])
   UNTIL j=0;
   FastSearch := LegalPairing(topgame,botgame)
END; {FastSearch}

FUNCTION PartialSearch (t,b: INTEGER): BOOLEAN;
{ PartialSearch tries all exchanges between players in the lower half
  of the score group, as they can be found after application of
  procedure FirstPairingAttempt in array Pairing[topgame..botgame, 2],
  and selects the cheapest one. }

VAR topgame,botgame,i,SumOfCosts,CheapSum: INTEGER;
    CurCost: ARRAY [1..MaxNrOfGames] OF INTEGER;
    Cheapest: ARRAY [1..MaxNrOfGames] OF CONTESTANT;

    PROCEDURE GenerateLowerHalfPermutations
              (tg,bg,level,CostBefore: INTEGER);
    ....

BEGIN
   topgame := (t+1) DIV 2;
   botgame := (b+1) DIV 2;
   FirstPairingAttempt(t,b,topgame,botgame);
   SumOfCosts:=0;
   FOR i:=topgame TO botgame
```

```
   DO BEGIN
      Cheapest[i]:= pairing[i,2];
      CurCost[i]:= Cost(pairing[i,1],pairing[i,2]);
      SumOfCosts:= SumOfCosts + CurCost[i]
   END; {OD}
   CheapSum:= SumOfCosts;

   GenerateLowerHalfPermutations(topgame,botgame,topgame,0);

   FOR i:=topgame TO botgame DO pairing[i,2]:= Cheapest[i];
   PartialSearch := LegalPairing(topgame,botgame)
END; {PartialSearch}

FUNCTION ExhaustiveSearch (t,b: INTEGER): BOOLEAN;
{ ExhaustiveSearch tries all exchanges between players in the score
  group, as they can be found after application of procedure
  FirstPairingAttempt in array Pairing[topgame..botgame, 1..2],
  and selects the cheapest one. }

VAR topgame,botgame,i,SumOfCosts,CheapSum: INTEGER;
    CurCost: ARRAY [1..MaxNrOfGames] OF INTEGER;
    Cheapest: ARRAY [1..MaxNrOfGames, 1..2] OF CONTESTANT;

    PROCEDURE GenerateScoreGroupPermutations
              (tg,bg,level,CostBefore: INTEGER);
    ....

BEGIN
   topgame := (t+1) DIV 2;
   botgame := (b+1) DIV 2;
   FirstPairingAttempt(t,b,topgame,botgame);

   SumOfCosts:=0;
   FOR i:=topgame TO botgame
   DO BEGIN
      Cheapest[i,1]:= pairing[i,1]; Cheapest[i,2]:= pairing[i,2];
      CurCost[i]:= Cost(pairing[i,1],pairing[i,2]);
      SumOfCosts:= SumOfCosts + CurCost[i]
   END; {OD}
   CheapSum:= SumOfCosts;

   GenerateScoreGroupPermutations(topgame,botgame,topgame,0);

   FOR i:=topgame TO botgame
   DO BEGIN
      pairing[i,1]:= Cheapest[i,1];
      pairing[i,2]:= Cheapest[i,2]
   END; {OD}
```

```
   ExhaustiveSearch := LegalPairing(topgame,botgame)
END; {ExhaustiveSearch}


PROCEDURE Forward (VAR t,b: INTEGER);
...
PROCEDURE AddGroup (VAR t,b: INTEGER);
...

{ Main pairing algorithm }

FUNCTION MakePairing : BOOLEAN;
VAR game,top,bottom: INTEGER;
    Success,SingleScoreGroup: BOOLEAN;
BEGIN
Success := FALSE; SingleScoreGroup:=TRUE;
FOR game :=1 TO (NrOfPlayersToBePaired) DIV 2
DO GroupBorders[game]:=FALSE;
top := 1; CurScore := score(players[1]);
bottom := 1;
WHILE (bottom < NrOfPlayersToBePaired)
      AND (score(players[bottom+1]) = curscore)
{this depends on lazy left to right evaluation of operator AND}
DO Inc(bottom);

WHILE (top <= NrOfPlayersToBePaired) DO
BEGIN

   GroupBorders[(top DIV 2) + 1]:= TRUE;

   { Depending on the number of players to be paired in a single score
     group, first a quick convergence algorithm is tried, weeding out
     the worst matches by exchanging them with others. }
   IF SingleScoreGroup AND (bottom - top > FastLimit)
   THEN success := FastSearch(top,bottom)
   ELSE success := FALSE;

   { If either the number of players in a single score group is too small,
     but not small enough to allow for a full search, or if the exchanges
     above fail to produce an acceptable solution, a backtrack over all
     possible permutations of the lower half is tried. }
   IF SingleScoreGroup AND (bottom - top > FullLimit) AND NOT success
   THEN success := PartialSearch(top,bottom);

   { Finally, usually only when the number of players is small enough
     to allow for a full search, but also if needed because partial
     search was not successful or if more than one score group must be
     paired, exhaustive search is applied. }
```

```
   IF NOT success
   THEN success := ExhaustiveSearch(top,bottom);

   IF success
   THEN BEGIN Forward(top,bottom); SingleScoreGroup := TRUE END
   ELSE BEGIN AddGroup(top,bottom); SingleScoreGroup := FALSE END;

END; {OD}

IF Success
THEN
 BEGIN
    Inc(LastPlayedRound);
    FOR Game := 1 TO (NrOfPlayersToBePaired-1) DIV 2
    DO BEGIN
       AllocateColors(Game);
       CreateGame(pairing[Game,2],pairing[Game,1],LastPlayedRound);
    END;
    Game:=(NrOfPlayersToBePaired+1) DIV 2;
    IF (pairing[Game,2] = Dummy)
    THEN SetExemption(pairing[Game,1],LastPlayedRound)
    ELSE BEGIN
       AllocateColors(Game);
       CreateGame(pairing[Game,2],pairing[Game,1],LastPlayedRound);
    END
 END;
MakePairing := Success
END; {MakePairing}
```

Bibliography

[ASM79] Abrial, J.-R., Schuman S.A., Meyer, B.: Specification language. In McKeag, R.M., MacNaughten, A.M. (eds.): *On the construction of programs*, Oxford University Press, 1979.

[Agr86] Agresti, W.M. (ed.): *New paradigms for software development.* IEEE Computer Society Press, Washington, D.C., 1986.

[Bac86] Backhouse, R.C.: *Program Construction and Verification.* Prentice-Hall, 1986.

[Bal81] Balzer, R.: "Final report on GIST". Technical Report USC/ISI, Marina del Rey, 1981.

[Bak84] Baker-Finch, C.: "Acceptable models of algebraic semantics," in C.J. Barter (ed.), Proceedings of the Seventh Australian Computer Science Conference, Adelaide, *Australian Computer Science Communications*, vol. 6, no. 1, pp. 5-1/10, 1984.

[Bau81] Bauer, F.L.: Programming as fulfillment of a contract. In Henderson, P. (ed.): *System design.* Infotech State of the Art Report, vol. 9, no. 6, pp. 165-174. Pergamon Infotech Ltd., Maidenhead, 1981.

[BBK92] Bernot, G., Bidoit, M., and Knapik, T.: "Towards an adequate notion of observation," in: B. Krieg-Brückner (ed.), *ESOP '92*, pp. 39-55, Lecture Notes in Computer Science, vol. 582, Springer, Berlin, 1992.

[BDMW81] Broy, M., Dosch, W., Möller, B., and Wirsing, M.: "GOTOs – a study in the algebraic specification of programming languages." In Brauer, W. (ed.): *Informatik-Fachberichte 50, GI – 11. Jahrestagung*, pp. 109-121, Springer, 1981.

[BCG83] Balzer, R., Cheatham, T.E. Jr., Green, C.: "Software technology in the 1990's: using a new paradigm," *Computer*, November 1983, pp. 39-45.

[BG79] Balzer, R., Goldman, N.: Principles of good software specification and their implications for specification languages. In: *Proc. Specifications of Reliable Software*, Cambridge, Mass., 1979.

[BEP87] Blum, E.K., Ehrig, H., and Parisi-Presicce, F.: "Algebraic specification of modules and their basic interconnections", *Journal of computer and system sciences*, vol. 34, pp. 293-339, 1987.

[BG80] Burstall, R.M., Goguen, J.A.: Semantics of CLEAR, a specification language. In Bjørner, D. (ed.): *Abstract software specifications*, pp. 292-332, Lecture Notes in Computer Science, vol. 86, Springer, Berlin, 1980.

[BHK89] Bergstra, J.A., Heering, J., and Klint, P. (eds.): *Algebraic specification*, ACM Press frontier series, Addison Wesley, New York, 1989.

[BHK90] Bergstra, J.A., Heering, J., and Klint, P.: "Module algebra", *Journal of the ACM*, vol. 37, no. 2, pp. 335-372, 1990.

[BMPP89] Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: "Programming by formal reasoning – computer-aided intuition-guided programming," *IEEE Transactions on Software Engineering*, vol. 15, no. 2, 1989.

[BK86] Bergstra, J.A., and Klop, J.W.: "Conditional rewrite rules: confluence and termination," *Journal of Computer and System Sciences*, vol. 32, no. 3, pp. 323-362, 1986.

[Bro87] Broy, M.: "Predicative specifications for functional programs describing communicating networks," *Information Processing Letters*, vol. 25, pp. 93-101, 1987.

[BT82] Bergstra, J.A., and Tucker, J.V.: "The completeness of the algebraic specification methods for computable data types," *Information and Control*, vol. 54, no. 3, pp. 186-200, 1982.

[BT83] Bergstra, J.A., and Tucker, J.V.: "Initial and final algebra semantics for data type specifications: two characterization theorems," *SIAM Journal on Computing*, vol. 12, no. 2, pp. 366-387, 1983.

[BW88] Bird, R.S., Wadler, P.L.: *Introduction to functional programming.* Prentice Hall, Hemel Hempstead, 1988.

[COMPASS91] Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., and Sanella, D. (eds.): *Algebraic System Specification and Development, A Survey and Annotated Bibliography.* Lecture notes in Computer Science, vol. 501, Springer, Berlin, 1991.

[DE84] Drosten, K., and Ehrich, H.-D.: "Translating algebraic specifications to Prolog programs," Informatik-Bericht Nr. 84-08, Technische Universität Braunschweig, 1984.

[Die88] Diepen, N.W.P. van: Implementation of Modular Algebraic Specifications (extended abstract). In H. Ganzinger (ed.): *ESOP '88*, pp. 64-78, Lecture Notes in Computer Science, vol. 300, Springer, Berlin, 1988.

[DHR88] Dubois, E., Hagelstein, J., and Rifaut, A.: "Formal requirements engineering with ERAE," *Philips Journal of Research*, vol. 43, no. 3/4, pp. 393-414, 1988.

[DP91] Diepen, N.W.P. van, and Partsch, H.A.: Formalizing Informal Requirements. Some Aspects. In L.M.G. Feijs, J.A. Bergstra, (eds.): *Algebraic Methods II: Theory, Tools and Applications*, pp. 7-27, Lecture Notes in Computer Science, vol. 490, Springer, Berlin, 1991.

[DR86] Diepen, N.W.P. van, and Roever, W.P. de: "Program derivation through transformations: the evolution of List-Copying Algorithms," *Science of Computer Programming*, vol. 6, pp. 213-272, 1986.

[Dij76] Dijkstra, E.W.: *A discipline of programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Ehl85] Ehler, H.: "Making formal specifications readable." Institut für Informatik, TU München, Report TUM-I8527, 1985.

[EFH83] Ehrig, H., Fey, W., and Hansen, H.: "ACT ONE - an algebraic specification language with two levels of semantics." TU Berlin, Technical Report 83-03, 1983.

[EM90] Ehrig, H., and Mahr, B.: *Fundamentals of Algebraic Specifications 2: Module Specifications and Constraints.* EATCS Monographs on Computer Science, vol. 21, Springer, Berlin, 1990.

[Fai85] Fairley, R.: *Software engineering concepts.* McGraw-Hill, New York, 1985.

[Fea86] Feather, M.S.: A survey and classification of some program transformation approaches and techniques. In Meertens, L.G.L.T. (ed.): *Program specification and transformation.* Proc. IFIP TC 2 Working Conference, Bad Tölz, April 15-17, 1986. North-Holland, Amsterdam, 1987.

[Fey86] Fey, W.: "Introduction to Algebraic Specification in ACT TWO." T.U. Berlin, Technical Report 86-13, 1986.

[FGJM85] Futatsugi, K., Goguen, J.A., Jouannaud, J.-P., and Meseguer, J.: Principles of OBJ2. In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 52-66, ACM, 1985.

[FJOKRR87] Feijs, L.M.G., Jonkers, H.B.M, Obbink, J.H., Koymans, C.P.J., Renardel de Lavalette, G.R., and Rodenburg, P.H.: A survey of the design language COLD. In: *ESPRIT '86: Results and Achievements*, pp. 631-644. North-Holland, Amsterdam, 1987.

[Gau85] Gaudel, M.C.: Toward structured algebraic specification. In: *ESPRIT '85: Status Report of Continuing Work. Part I*, pp. 493-510. North-Holland, Amsterdam, 1986.

[Ger94] Gerlach, C.: "Ein MacMahon-Losungsprogramm für Go-Turniere unter Benutzung von Maximum Weight Perfect Matching." Master's Thesis, Universität Hildesheim, 1994 (in German).

[GH83] Guttag, J.V., and Horning, J.J.: "Preliminary Report on the LARCH shared language." Technical Report CSL 83-6, Xerox, Palo Alto, 1983.

[GH88] Gijssen, G., and Haggenburg, W.G.: *Zwitsers Systeem.* KNSB, Amsterdam, 1988 (partially in Dutch).

[GM81] Goguen, J., and Meseguer, J.: "OBJ-1, a study in executable algebraic formal specifications." SRI International Technical Report, 1981.

[GM82] Goguen, J.A., and Meseguer, J.: Universal realization, persistent interconnection and implementation of abstract modules. In Nielsen, M., and Schmidt, E.M. (eds.): *Proceedings 9th International Conference on Automata, Languages and Programming*, pp. 265-281, Lecture Notes in Computer Science, vol. 140, Springer, 1982.

[GM84] Goguen, J.A., and Meseguer, J.: "Equality, types, modules, and (why not?) generics for logic programming," *Journal of Logic Programming*, vol. 2, pp. 179-210, 1984.

[GMP83] Goguen, J.A., Meseguer, J., and Plaisted, D.: Programming with parameterized abstract objects in OBJ. In Ferrari, D., Bolognani, M., and Goguen, J.A.: *Theory and Practice of Software Technology*, pp. 163-193, North-Holland, 1983.

[Gor79] Gordon, M.J.C.: *The denotational description of programming languages.* Springer, New York, 1979.

[Gri81] Gries, D.: *The science of programming.* Springer, Berlin, 1981.

[GT77] Goguen, J.A., and Tardo, J.: "OBJ-0 preliminary users manual." University of California at Los Angeles, Computer Science Department, 1977.

[GT93] Bleeker, A., Boudewijns, M., Evers, M., Scholten, C., Schoonderbeek, J., Toonen, P., and Wiggerts, Th.: *GoMMTour User Manual v. 2.03*, Nijmegen, 1993.

[Hen80] Henderson, P.: *Functional programming: application and implementation.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[Hen81] Henderson, P.: System design: analysis. In: *System design,* pp. 5-163, Infotech State of the Art Report, vol. 9, no. 6, Pergamon Infotech Ltd., Maidenhead, 1981.

[Hen91] Hennicker, R.: "Observational implementation of algebraic specifications," *Acta Informatica*, vol. 28, pp. 187-230, 1991.

[Her88] Herik, J. van den: "Computerschaak," *Schakend Nederland*, vol. 95, no. 9, pp. 38-39, 1988 (in Dutch).

[Her91] Herik, J. van den: "Computerschaak – Zwitsers Systeem," *Schakend Nederland*, vol. 98, no. 11, pp. 35-36, 1991 (in Dutch).

[HGM86] Hehner, E.C.R., Gupta, L.E., and Malton, A.J.: "Predicative Methodology," *Acta Informatica,* vol. 23, pp. 487-505, 1986.

[HHKR89] Heering, J., Hendriks, P.R.H., Klint, P., and Rekers, J: "The syntax definition formalism SDF – reference manual," *SIGPLAN Notices*, vol. 24, no. 11, pp. 43-75, November 1989.

[HO80] Huet, G., and Oppen, D.C.: Equations and rewrite rules: a survey. In R.V. Book (ed.): *Formal Language Theory, Perspectives and Open Problems*, pp. 349-405, Academic Press, 1980.

[Huß87] Hußmann, H.: "RAP-2 User Manual." Universität Passau, Fachbereich Mathematik und Informatik, Technical Report, 1987.

[IEEE77] *Special Collection on Requirement Analysis.* IEEE Transactions on Software Engineering SE-3:1, pp. 2-84, 1977.

[IEEE83] *IEEE Standard Glossary of software engineering terminology.* IEEE Standard 729, 1983.

[JKR86] Jonkers, H.B.M., Koymans C.P.J., and Renardel de Lavalette, G.R.: "A semantic framework for the COLD-family of languages." Logic Group Preprint Series No. 9, Department of Philosophy, University of Utrecht, 1986.

[Jon80] Jones, C.B.: *Software Development: a Rigorous Approach.* Prentice-Hall, 1980.

[JW78] Jensen, K., and Wirth, N.: *Pascal: User Manual and Report* (second edition). Springer, 1978.

[Kam83] Kamin, S.: "Final data types and their specification," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 97-123, 1983.

[Kaž80] Kažić, B.M.: *The Chess Competitor's Handbook.* Badsford, London, 1980.

[KK91] Kaliski, M.E., and Kaliski, B.S.: *The Software Sleuth.* West Publishing, St. Paul, 1991.

[KPR87] Kühnel, B., Partsch, H., and Reinshagen, K.P.: "Requirements Engineering — Versuch einer Begriffsklärung," *Informatik-Spektrum*, vol. 10, no. 6, pp. 334-335, 1987 (in German).

[Leh80] Lehman, M.M.: "Programs, life cycles, and laws of software evolution," *Proc. IEEE,* vol. 68, no. 9, 1980.

[LF82] London, P., and Feather, M.S.: "Implementing specification freedom," *Science of Computer Programming*, vol. 2, pp. 91-131, 1982.

[LS84] Loeckx, J., and Sieber, K.: *The Foundation of Program Verification.* Wiley-Teubner, 1984.

[MG85] Meseguer, J., and Goguen, J.A.: Initiality, induction, and computability. In Nivat, M., and Reynolds, J.C. (eds.): *Algebraic Methods in Semantics*, pp. 459-541, Cambridge University Press, 1985.

[Möl87] Möller, B.: *Higher-order algebraic specifications.* Habilitation thesis, Fakultät für Mathematik und Informatik, T.U. München, 1987.

[MTW88] Möller, B., Tarlecki, A., and Wirsing, M.: Algebraic specification with built-in domain constructions. In Dauchet, M., Nivat, M. (eds.): *CAAP '88*, pp. 132-148, Lecture Notes in Computer Science, vol. 299, Springer, Berlin, 1988.

[ODo85] O'Donnell, M.J.: *Equational Logic as a Programming Language.* MIT Press, 1985.

[Par86] Partsch, H.: "Algebraic requirements definition: a case study," *Technology and Science of Informatics,* vol. 5, no. 1, pp. 21-36, 1986.

[Par87] Partsch, H.: Requirements Engineering und Formalisierung — Problematik, Ansatz und erste Erfahrungen. In Schmitz, P., Timm, M., Windfuhr, M. (eds.): *Requirements Engineering '87*, pp. 9-31. GMD-Studien 121, 1987 (in German).

[Par89] Partsch, H.: Algebraic specification — A step towards future software engineering. In Wirsing, M., and Bergstra, J.A., (eds.): *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Computer Science, vol. 394, Springer, Berlin, 1989.

[Par90] Partsch, H.A.: *Specification and Transformation of Programs, A formal approach to software development.* Texts and Monographs in Computer Science, Springer, Berlin, 1990.

[PL82] Partsch, H., and Laut, A.: From requirements to their formalization — a case study on the stepwise development of algebraic specifications. In Wössner, H. (ed.): *Programmiersprachen und Programmentwicklung*, 7. Fachtagung, München 1982, pp. 117-132. Informatik-Fachberichte 53, Springer, Berlin, 1982.

[PS82] Papadimitriou, C.H., and Steiglitz, K.: *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, 1982.

[PS83] Partsch, H., and Steinbrüggen, R.: "Program transformation systems," *ACM Computing Surveys*, vol. 15, pp. 199-236, 1983.

[PTW83] Polya, G., Tarjan, R.E., and Woods, D.R., *Notes on Introductory Combinatorics.* Basel, 1983.

[Rom85] Roman, G.-C.: "A taxonomy of current issues in requirements engineering," *IEEE Computer*, vol. 18, no. 4, pp. 14-23, 1985.

[RO85] Rzepka, W.E., Ohno, Y.: "Requirements engineering environments: software tools for modeling user needs," *IEEE Computer*, vol. 18, no. 4, pp. 9-12, 1985.

[Sch87] Schoett, O.: *Data abstraction and the correctness of modular programming.* Ph. D. Thesis, Department of Computer Science, University of Edinburgh, Report CST-42-87, 1987.

[SL89] Smith, D.R., and Lowry, M.J.: Algorithm theories and design tactics. In Snepscheut, J.L.A. van de (ed.): *Proc. Mathematics of Program Construction*, pp. 379-398, Lecture Notes in Computer Science, vol. 375, Springer, Berlin, 1989.

[ST87] Sannella, D., and Tarlecki, A.: "On observational equivalence and algebraic specification," *Journal of Computer and System Sciences*, vol. 34, pp. 150-178, 1987.

[ST88] Sannella, D., and Tarlecki, A.: "Towards formal development of programs from algebraic specifications: implementations revisited," *Acta Informatica*, vol. 25, pp. 233-281, 1988.

[Swa82] Swartout, W.: GIST English generator. In: *Proc. AAAI 82*, August 1982.

[Thi94] Thienen, H. van: *It's about time – using Funmath for the description and analysis of discrete dynamic systems.* Ph.D. Thesis, Katholieke Universiteit Nijmegen, 1994.

[Wal91] Walters, H.R.: *On Equal Terms - Implementing Algebraic Specifications.* Ph.D. Thesis, Universiteit van Amsterdam, 1991.

[Wan79] Wand, M.: "Final algebra semantics and data type extensions," *Journal of Computer and System Sciences*, vol. 19, pp. 27-44, 1979.

[Web74] *Webster's New World Dictionary.* Second College Edition. William Collings & World Publishing, Cleveland, 1974.

[Wir83] Wirsing, M.: *A Specification Language.* Habilitation thesis, Fachbereich Mathematik und Informatik, T.U. München, 1983.

[Yeh82] Yeh, R.T.: Requirements analysis — a management perspective. In: *Proc. COMPSAC '82*, pp. 410-416, November 1982.

[YZ80] Yeh, R.T., and Zave, P.: "Specifying software requirements," *Proc. IEEE*, vol. 68, no. 9, pp. 1077-1085, 1980.

Summary

The subject of this Thesis is the production of correct and efficient software in a structured way. Obviously, the complexity of the problems to be solved by software is large enough to warrant a systematic approach. Current methods have failed to produce correct, let alone efficient, software on a regular basis. Now, as for the past 25 years, software is typically late, suffering from 'bugs' (an euphemism for 'faults'), and its performance is not according to expectations, if it is delivered at all.

This Thesis provides one way to tackle the complexity of software production. Its tools are:

program transformations One way to write a correct and efficient program is to start with an evidently correct program, or even a non-executable specification. Next, this program is transformed using correctness-preserving steps to another program (which may be more efficient). Programs developed in this way are efficient if the transformations are chosen adequately. At the same time they are correct by construction, so a separate correctness proof is not necessary.

modular algebraic specifications Unfortunately current techniques for program transformations are not well-suited to large programs. So either improvement of the techniques is needed, or programs are divided into smaller modules of the right size. In this thesis the latter route is chosen.

The choice for the specification formalism used is not very well-defined. Algebraic specifications are convenient. They are semantically close to traditional mathematics, and hence easy to understand intuitively. And they can be used as a starting point for program transformations and modularization, the necessary attributes in this Thesis. However, most techniques in the Thesis are not bound to algebraic specifications only, and can be transported to other modular specification formalisms.

Chapter 1 is an introduction to the formalisms used, especially modular algebraic specifications, and it gives some necessary background for the technique of program transformations. It also provides a summary of the conclusions and recommendations. The rest of the Thesis closely follows the normal course of formal program development.

Chapters 2 and 3 deal with formal specifications, in this case algebraic specifications. In Chapter 2 it is argued that writing specifications is a non-trivial task in itself. This is illustrated using an example from real life: the Swiss System. Even

though a good description exists and a pseudo-algorithm suitable for human application is provided in the informal specification, a formal specification is hard to come by. The next Chapter provides a more happy note: it gives a rather elegant formal algebraic specification of the **goto**-statement, notoriously the most difficult to specify among the classical program constructs.

The heart of the Thesis can be found in Chapter 4. The modularization constructs provided by modular algebraic specifications are augmented with a few rules-of-conduct. Those rules are quite natural, and their application can be easily checked by computer. They allow the decomposition of algebraic specification in sufficiently small modules. Each module can be observed via a limited number of functions. Only these functions must be implemented according to the specification, hence they can be implemented separately, using currently available transformational techniques. Other functions remain sufficiently behind-the-screen to allow for optimizing implementations.

The final Chapter shows the derivation of a difficult algorithm (the Swiss System specified in Chapter 2). This problem can be translated into terms of Combinatorics. However, current Combinatorial algorithms produce correct, but unacceptably slow (at best $\mathcal{O}(n^3)$ – n the number of players) software. Hence the specification is relaxed, using domain knowledge, to derive an algorithm which is linear on average.

The Thesis shows the following results:

- The process of formalization is difficult. Upon reflection even precise statements in natural language are often surprisingly imprecise. Hence automated support is a necessity.

- Modular algebraic specifications are a powerful and flexible specification mechanism, as shown with two original example specifications: the Swiss System and the **goto**-statement.

- Observational semantics of modular algebraic specifications are introduced. This allows for separate implementation of modules, thus making the application of transformational techniques, local optimization, and program reuse possible.

  The extra proof obligations necessary to allow the application of separate implementations are shown to be simple syntactical checks in most practical cases.

- The power of transformational derivations is shown through derivations at the specification level, introducing domain knowledge at an early stage. This made a new and efficient implementation of the Swiss System possible, improving on the usual Combinatorical solution.

Samenvatting

Het onderwerp van dit proefschrift is het gestructureerd produceren van correcte en efficiënte software. Het is duidelijk dat de complexiteit van de problemen die met software moeten worden opgelost groot genoeg zijn om een systematische aanpak noodzakelijk te maken. De nu gangbare methoden hebben niet geleid tot geregelde productie van correcte, om maar te zwijgen van efficiënte, software. Nog steeds, net als de laatste 25 jaar, is software meestal te laat, lijdend aan 'bugs' ('luizen', een eufemisme voor 'fouten'), en is het gedrag niet wat er van verwacht wordt, als het al wordt afgeleverd.

Dit proefschrift voorziet in een manier om de complexiteit van de productie van software aan te vatten. De gehanteerde gereedschappen zijn:

programma-transformaties Eén manier om correcte en efficiënte programma's te schrijven begint met een evident correct programma, of zelfs met een niet-executeerbare specificatie. Vervolgens wordt dit programma met de correctheid behoudende stappen getransformeerd in een ander programma (dat al dan niet efficiënter is). Een op deze manier ontwikkeld programma is efficient als de transformaties goed gekozen zijn. Tegelijkertijd is het programma door de manier van construeren correct gebleven, dus een apart correctheidsbewijs kan achterwege blijven.

modulaire algebraïsche specificaties Helaas zijn de nu gebruikelijke technieken voor programmatransformatie niet erg geschikt voor het toepassen op grote programma's. Er is dus een verbetering van die technieken nodig, òf programma's moeten worden verdeeld in kleinere modulen van de juiste maat. De laatste weg wordt in dit proefschrift genomen.

De keuze voor het specificatieformalisme is niet zo goed onderbouwd. Algebraïsche specificaties voldoen. Hun semantiek ligt dicht bij de traditionele wiskunde, dus ze zijn tamelijk eenvoudig intuïtief te begrijpen. Ook kan zo'n specificatie als beginpunt voor programmatransformaties en voor modularisatie dienen, de in dit proefschrift noodzakelijke eigenschappen. Maar de meeste technieken in dit proefschrift zijn niet beperkt tot algebraïsche specificaties, en ze kunnen overgezet worden op andere modulaire specificatieformalismen.

Hoofdstuk 1 introduceert de gebruikte formalismen, in het bijzonder modulaire algebraïsche specificaties, en het voorziet in de noodzakelijke achtergrond bij de techniek van programmatransformaties. Ook bevat het een samenvatting van de

conclusies en aanbevelingen. De rest van het proefschrift volgt de normale lijn van formele programma-ontwikkeling.

De hoofdstukken 2 en 3 behandelen formele specificaties, in dit geval algebraïsche specificaties. Het schrijven van specificaties is geen eenvoudige zaak, zoals hoofdstuk 2 laat zien. Ter illustratie is er een practisch voorbeeld opgenomen: het Zwitsers systeem. Hoewel er een goede beschrijving bestaat en er een pseudo-algoritme voor menselijke toepassing beschikbaar is als informele specificatie, is het heel lastig een formele specificatie te maken. Het volgende hoofdstuk zet een vrolijker toon met een tamelijk elegante formele specificatie van het **goto**-statement (de sprongopdracht), notoir de lastigste klassieke programmabouwsteen om te specificeren.

Het centrale deel van het proefschrift staat in hoofdstuk 4. De modularisatie-constructiemechanismen die bij modulaire algebraïsche specificaties horen worden aangevuld met een paar gedragsregels. Deze vuistregels zijn tamelijk natuurlijk, en ook eenvoudig door de computer te controleren. Daarmee kunnen algebraïsche specificaties dan in voldoende kleine modulen worden onderverdeeld. Elk van deze modulen kan worden geobserveerd met behulp van een beperkte verzameling van de bijgehorende functies. Alleen de functies in deze verzameling moeten precies volgens de specificaties worden geïmplementeerd, dus de implementaties kunnen elk apart worden gemaakt met behulp van de nu beschikbare transformatietechnieken. De andere functies blijven daarbij voldoende uit het zicht om geoptimaliseerde implementaties mogelijk te maken.

Het afsluitend hoofdstuk geeft een afleiding van een moeilijk algoritme (het Zwitsers systeem uit hoofdstuk 2). Dit probleem kan worden vertaald naar de combinatoriek, maar de huidige combinatorische algoritmen produceren weliswaar correcte, maar onacceptabel langzame (niet beter dan $\mathcal{O}(n^3)$ met n het aantal spelers) programmatuur. De specificatie wordt daarom verzwakt waarbij van domeinkennis gebruik wordt gemaakt, waarna een gemiddeld lineair algoritme kan worden afgeleid.

Het proefschrift bevat de volgende resultaten:

- Het formalisatieproces is moeilijk. Bij nadere beschouwing blijken zelfs precieze uitspraken in natuurlijke taal verrassend ambigu te zijn. Er is dus duidelijk behoefte aan automatische ondersteuning.

- Modulaire algebraïsche specificaties zijn een krachtig en flexibel specificatiemechanisme. Dit wordt door twee nieuwe voorbeelden geïllustreerd: het Zwitsers systeem, en de sprong-opdracht.

- Voor modulaire algebraïsche specificaties wordt observationele semantiek geïntroduceerd. Dit maakt gescheiden implementatie van modulen mogelijk, en daarmee de toepassing van transformatietechnieken, lokale optimalisatie, en hergebruik van programmatuur.

  De voor gescheiden implementatie noodzakelijke extra bewijsverplichtingen blijken voor de praktijk meestal op eenvoudige syntactische testen terug te brengen.

- De kracht van transformationele afleidingen wordt getoond door het toepassen van afleidingen op specificatie-niveau, waarbij al vroeg van domeinkennis ge-

bruik wordt gemaakt. Hiermee is een nieuwe, vergeleken met de standaardoplossing uit de combinatoriek efficiënte, implementatie van het Zwitsers systeem gegeven.

Curriculum Vitae

1959	Geboren op 12 maart te Wormer (N.H.)
1971-1973	1e en 2e klas, Gymnasium Juvenaat H. Hart (Bergen op Zoom)
1973-1974	3e klas Gymnasium, Mencia de Mendoza-Lyceum (Breda)
1974-1977	4e t/m 6e klas Athenaeum B, O.L. Vrouwe-Lyceum (Breda)
juni 1977	eindexamen V.W.O.
1977-1984	studie Wiskunde met groot bijvak Informatica, Rijksuniversiteit Utrecht
1984	didactische aantekening Wiskunde 1e graads
januari 1985	Doctoraalexamen wiskunde met bijvak Informatica
1985-1988	Wetenschappelijk Onderzoeker Centrum voor Wiskunde en Informatica, Amsterdam (ESPRIT-project Generation of Interactive Programming Environments – GIPE)
1988-1991	Wetenschappelijk Onderzoeker Katholieke Universiteit Nijmegen (NWO-project Specification and Transformation Of Programs – STOP)
1992-heden	Universitair Docent Vakgroep Informatica Katholieke Universiteit Nijmegen