# Reductivity Arguments and Program Construction

Doornbos, Henk
Reductivity Arguments and Program Construction / Henk Doornbos. -
Eindhoven: Eindhoven University of Technology
Thesis Technische Universiteit Eindhoven. -
With index, ref. - With summary in Dutch
ISBN 90-386-0098-4
Subject headings: relation calculus, transformational programming.

druk: Universitaire Drukkerij, Eindhoven


*The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).*

# Reductivity Arguments and Program Construction

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op maandag 24 juni 1996 om 16.00 uur

door

**Hendrik Doornbos**

geboren te Ten Post

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. R. C. Backhouse
en
prof.dr. E. W. Dijkstra

# Acknowledgements

Many people have contributed in making the research reported in this thesis possible and enjoyable. I would like to mention:

- My Ph.D. supervisor Roland Backhouse, for accepting me as a Ph.D. student, showing me his good taste, teaching me that it is always possible to make things better and helping me in doing so.

- My fellow Ph.D. students Paul Hoogendijk and Ed Voermans, for their enthusiasm and interest.

- The other members and former members of the Eindhoven Mathematics of Program Construction Group: Chritiene Aarts, Eerke Boiten, Joop van den Eijnde , Netty van Gasteren, Rik van Geldrop, Frans Rietman, and Jaap van der Woude.

- The external members of the Ph.D. committee: Edsger W. Dijkstra, Wim Hesselink and Anne Kaldewaij, for reviewing this thesis and giving detailed comments.

- Ronald Bulterman, Carel Scholten, and other members of the Eindhoven Tuesday Afternoon Club, and Rutger Dijkstra and Burghard von Karger for their interest in my work on well-foundedness.

- Wim Feijen, who taught me many things and gave me the final kick; I owe him a lot.

- Netty van Gasteren, who helped me writing in spite of local circumstances.

- Oege de Moor, for the inspiration for the notion of reductivity.

- Laurens de Vries and Judith Masthoff, for the many pleasant and fruitful conversations.

- Richard Verhoeven, for implementing the Math∫pad system, and providing LaTeX help and other assistance.

- My family and friends, to whom I apologise for not wanting to tell them about my work.

# Contents

# Chapter 1

# Introduction

## 1.1 Aims and motivation

We are programmers, in the sense that it is our concern to improve the process of program construction. Therefore we want to answer questions like: What is programming, why is it so difficult and error-prone, and how can we learn what is needed to make the process more manageable?

The contribution of this thesis to the improvement of programming is that it tries to identify some of the concepts that play an important role in the construction of programs and seeks to formalise them in such a way that they can be used in practice. Two such concepts are, of course, specifications and programs.

We limit our attention to just one phase of the programming process: for us programming is the construction of programs from specifications. Of course, that does not mean that we consider other aspects of programming unimportant. For instance the phase of obtaining a specification from a vague description is well-known to be a major source of errors in the "final" product. It is not easy to obtain a "valid" specification, that is, one that captures exactly what the client has in mind and that does not contain any inconsistencies. Therefore, an important property of specifications is that they can be validated effectively. In this thesis however, it is assumed that valid specifications can be obtained and the question of what their properties should be such that they can be validated is ignored.

Instead, we seek an answer to the following question. How should one model specifications and programs in such a way that the program can be obtained from the specification by means of formal reasoning?

### 1.1.1 Precision and concision

The idea of formal reasoning — by which we mean the manipulation of uninterpreted formulae according to prescribed syntactic rules — seems to split the computing community into two distinct and opposing schools. There are the enthusiasts who fervently advocate its use, arguing its effectiveness and reliability, and there are the sceptics who dismiss it, arguing that it ignores the creative process in the discovery of new facts or the design of new systems. Whilst ourselves belonging very much to the enthusiasts we are nevertheless of the opinion that, at this point in time, the sceptics can muster much bigger artillery than we enthusiasts. There are two problems. First, the formal methods community is too concerned with the issue of (a posteriori) *verification* of software rather than harnessing formal methods to the task of its *construction*. Second, it is not sufficiently recognised that formal methods must combine *precision* with *concision*. Too often formal systems are large and complex, involving complex rules with large numbers of parameters, and even the simplest specifications stretch over several pages of text. Like programming languages of old, formal systems of today too often belong to the problem domain rather than the solution domain.

Induction illustrates the issue well: An inductive proof typically involves a creative step, namely the invention of the inductive hypothesis. There then follows a verification according to well-defined (and well-known) mathematical principles. Formal reasoning is undoubtedly effective in the verification step, but in most cases it plays a very subordinate rôle (if any at all) in the creative step. Formal reasoning is a service industry and not a production industry.

Or is it? We would argue that formal reasoning *can* contribute significantly to the creative aspects of computing (and mathematics in general) if much more emphasis is given to the search for crisp and compact (but of course still precise) formulations of the fundamental concepts we use in our everyday work as programmers. And not only should the formulations be compact and precise, they must also be such that they can be *used* effectively, that is it must be possible to reason with them concisely and precisely.

The topic of this thesis then is the search for such formulations. But first we have to find the right concepts to formalise: a somewhat unfortunate formalisation of the right concept is to be preferred over an excellent formalisation of a concept which is irrelevant to the process of programming.

### 1.1.2 Specifications

To start the search for relevant concepts we recall our view of programming: the construction of programs from formal specifications. To be able to discuss programming as we see it we therefore need to have an idea of what a specification is. Basically, most formal specification methods such as Z and VDM have in common that in some formalism an input-output relation is specified. (Of course other aspects can be involved such as the required efficiency of a solution.) That a specification is a relation rather than a function is important: a functional specification might be too specific. For example, in optimisation problems an optimum need not be unique and the specification should not mention which particular optimum has to be found. That could leave the programmer too little freedom to construct an efficient program. So, we take the point of view that a specification is a relation, this giving us the possibility to abstract away from the formalism in which it is expressed.

### 1.1.3 Programs

We also need to model programs. This topic belongs to the field known as semantics. We do not study semantics principally in order to prove compilers correct or to verify that programs do what they should do, let alone to show that models of programs do exist. Our interest is in how semantics can help in program construction. That this is, indeed, possible is evidenced by the work done in the field called axiomatic semantics.

Traditionally, in semantics programs are modelled and one concentrates on the mapping between programs and their models. Another issue that is considered to be important in semantics is whether or not appropriate models do exist. As said before, we do not address these issues; instead we concentrate on the properties of the model. This is motivated by the following. Ideally the model should contain only those aspects of programs that are relevant in the process of programming. It should abstract from the features of programming languages that are irrelevant for this (for instance the particular syntax). Furthermore, the model should have properties that support programming. Of course that does not mean that the mapping between concrete and abstract programs is not important. Requirement number one is that it be simple.

### 1.1.4 Design criteria for a model of programs

What then should such a model of programs be like? Programs and specifications have much in common and indeed there are a lot of formalisms in which programs are viewed as special cases of specifications. However, the two are not really the same. One aspect of a program is its input-output behaviour, the responsibility of the computer being to construct this. However a program also has properties like efficiency —which is connected with its syntactic form— and termination, which make no sense for specifications (although a specification can *mention* termination this does not give us a specification with the *property* termination). In this thesis we will in the first place study programs that terminate. It will turn out that all three aspects of programs mentioned above (input-output behaviour, termination and efficiency) can be assigned to equations in relations. Therefore, we advocate the view that programs are equations in relations.

### 1.1.5 Datatypes

A third concept to be discussed besides programs and specifications are datatypes. In programming they play an important role. First, it has long been observed that data structure influences program structure. Second, efficient implementation of programs often requires the design of ingenious datatypes.

Traditionally a datatype is a set with a number of operations on it —an *algebra* in mathematical terminology. In this thesis we concentrate on the class of datatypes known as initial algebras because primarily these are known to influence program structure. The class is rather wide: it encompasses the natural numbers, lists, and also various tree-like structures. A set can be modelled as a relation and, of course, so can the operations on it, and that is how we do it in this thesis: datatypes are, like specifications, relations.

### 1.1.6 Relation calculus in programming

Now that we have identified three concepts that play an important role in programming, (specifications, programs, and datatypes) the question remains if this is enough to study at least part of the programming process. For us an important way of obtaining programs from specifications is by calculation, because calculation offers the possibility of isolating the straightforward parts of the programming process. Programming can be viewed as calculating an equation (the program) that is satisfied by a relation (the specification). In other words programming is calculating with relations. Therefore, the relation calculus is potentially a useful tool in programming and this is the motivation why the thesis uses this calculus as the theoretical framework.

An advantage of the relation calculus over, for instance, the predicate calculus is that it is point-free: it is not necessary to mention the points between which the relation holds. As a consequence formulae expressed in relation calculus tend to be very compact. And this in turn greatly facilitates effective calculational reasoning. But of course sometimes the points, which are the input and output of the program, do play a role. An important topic of investigation is therefore when point-free reasoning, using the relation calculus, is appropriate and when it is not and another calculus should be used. This question will be discussed briefly but is not a central one.

### 1.1.7 Termination

As we have argued before, the central topic of this thesis is the search for compact formulations of relevant concepts we use in our everyday work as programmers. In the previous sections we identified a number of these concepts, but of course there are more. Two studied in this thesis are termination and induction. To prove termination, and also other desirable properties, of a program typically an inductive argument is used, so it seems good to try to formalise these too.

It may seem obvious to formalise the concept termination, but in related work one has concentrated on another property of programs, namely that they have unique solutions. (Recall that in our view programs are equations and it therefore makes sense to talk about solutions of a program.) This was inspired by categorical approaches to semantics and the observation that in that framework unicity of solutions coincides with termination. A drawback of that approach is, however, that it only allows the study of a very limited class of programs, basically only those that can be defined by structural induction over their input. Such programs have been dubbed catamorphisms [62]. An example of such a program is the one that computes the sum of a list. But many functions do not fit comfortably into this class —the factorial function for example. It is even the case that there are algorithms that do not fit into the class at all. Programs defined by structural recursion are necessarily terminating. It is well-known that, consequently, there exist algorithms that are outside the class. ( Martin-Löf's theory of types [64] has the same drawback. It allows only programs defined by primitive recursion, and also all those programs are necessarily terminating.)

To overcome the limitations of structural recursion there was an explosion of so called morphisms: apart from catamorphisms we got paramorphisms [68], anamorphisms [69], zygomorphisms [62], hylomorphisms, prepromorphisms, postpromorphisms [42] and so on. All of these are classes of equations representing certain recursion schemes. One could then show that all these equations had unique solutions. However the technique to do so still relied on the categorical approach and the class of programs that could be tackled was still considerably smaller than the one a programmer has to work with.

This is an indication that unicity of solution is not the right concept to formalise and a contribution of this thesis is to formalise the notion of termination instead. This allows us to considerably extend the class of programs that can be dealt with, while simultaneously reducing the class of morphisms to a single member: the hylomorphism. The class of programs seems now large enough to tackle most, if not all, programming problems because it also contains the repetition. For instance, the linear search program which poses difficulties in the categorical approach fits comfortably in our approach.

### 1.1.8 Reductivity arguments and termination

At this point it is appropriate to explain the term "reductivity arguments" from the title of the thesis. What we mean by a reductivity argument could also be called an inductive argument. In this thesis we generalise what it means for a relation to admit induction, that is, when a relation can be used in an inductive argument. This is done by introducing a datatype in the definition. The generalisation is, however, not the same as the existing notion of "inductive relation" and therefore another term had to be invented.

It is well-known that a relation can be used in an inductive argument if and only if it is well-founded. Our generalisation is exemplified by the following. A relation between natural numbers and lists of natural numbers is reductive with respect to the datatype of lists if it is such that whenever it holds between a natural number $n$ and a list $m$ all the elements of list $m$ are smaller than the number $n$. In this sense the relation reduces its argument and this is the reason why the term "reductive" was chosen.

This new concept of reductivity is used to define what it means for a program to terminate and a proof of termination is therefore by a reductivity argument. But also all kinds of other arguments in which well-founded (or inductive) relations play a role can now be called reductivity arguments.

A major topic of this thesis is to develop a calculus of reductive relations. We do this by first giving a base class of reductive relations and then a number of methods to construct new reductive relations from old ones. This calculus then gives the possibility to reduce the proofs of termination to simple calculations led by syntactic criteria. It should also be possible to use the calculus for the construction of terminating programs.

## 1.2 Related work

### 1.2.1 Algorithmics

The idea of transforming specifications into programs, or transformational programming is actually quite old. For instance it was already mentioned in 1969 by Burstall and Landin [27]. Other, more well-known, references are Backus's Turing Award Lecture [13] and the "fold-unfold" method of Burstall and Darlington [26]. It is also the idea behind Iverson's work on the programming language APL [56], which also emphasises concision.

Transformational programming in the sense of algebraic manipulation of formulae, was further developed in the so-called Bird-Meertens formalism [21, 22]. Meertens calls the programming method algorithmics [67] to stress the fact that it is calculating with algorithms (like arithmetic, which is calculating with numbers). The original Bird-Meertens formalism was restricted to functions on the datatype of lists. Malcolm [63] subsequently showed how category theory could be used to introduce other datatypes. This line of research was further followed and developed by Fokkinga [42].

The introduction of non-determinacy is due to Backhouse et al [9, 8]. They proposed modelling programs (and also datatypes) as relations and to use relation calculus as the tool to transform programs. This calculus of binary relations was developed in the 19th century by Schröder and De Morgan and studied again in the 1940's and 50's by Tarski and his collaborators [82]. (See [60] for an account of the history of the relation calculus.) The contribution of Backhouse was to introduce datatypes in this calculus.

A comparable line of research has been set out by De Moor and Bird [17, 19, 18, 20]. However their approach differs from Backhouse's in that they choose to work in a categorical framework, based on Freyd's theory of allegories [44]. Although different in the underlying theory, the results of Bird and De Moor, and their way of calculating programs, are nearly the same as Backhouse's method. Because we are mainly interested in those aspects we choose to use the relation calculus, instead of a categorical approach, because it is somewhat simpler.

### 1.2.2 Refinement calculi

A separate line of research in transformational programming is based on the predicate transformer approach to semantics [35, 52]. In this approach programs are not viewed as functions from an initial state to a final state but as predicate transformers or, equivalently, transformers from sets of final states to sets of initial states. This method of semantics has proven to be very useful in the construction of programs [46, 36, 7, 57, 30]. A drawback could be that it is not customary to calculate with the programs themselves. Instead, one calculates with the programs applied to a predicate, i.e. with predicates. To overcome this, Back and von Wright [5] and Morgan [71] define a refinement relation on programs, expressing that one program is an implementation of another. Also, a new class of programs is introduced to model specifications. This way it becomes possible to transform a specification into a program by means of a calculation with programs.

It is well-known that the predicate transformer approach to program semantics is equivalent to a relational semantics. Therefore, the work of Backhouse et al. on a relational approach to algorithmics can be viewed as a bridge between algorithmics and refinement calculi.

### 1.2.3 Our contribution

The work in this thesis can be seen as a further contribution to the unification of the two streams of research on transformational programming. We extend the work of Backhouse et al. by explicitly introducing the notion of termination in the calculus, something which is also present in the refinement calculi. Another new feature of our approach to algorithmics is that programs are not considered as essentially the same as specifications. A further new extension is the introduction of set transformers into the relation calculus. This way, it becomes possible to incorporate the work done on predicate transformer semantics and refinement calculi.

Algorithmics has concentrated on the calculation of functional programs, whereas the various refinement calculi are meant to derive imperative programs. Because we try to integrate the two approaches to transformational programming, some of the work done in this thesis can also be seen as a step towards the integration of the functional and imperative styles of programming

## 1.3 Overview of the thesis

Although the relation calculus has a long history one can not assume that readers are familiar with it. Therefore, Chapter 2 contains an introduction to this calculus. The axioms of the calculus are given and we present a carefully selected set of properties that are needed in the remainder of the text. To help readers who are not familiar with the calculus we relate the introduced operators to the programming concepts which they are intended to model. This way the reader can convince himself of the validity of the given properties and judge whether or not they are useful for him. In doing so, the chapter also explains our view on specifications, programs and datatypes.

An important part of our approach is the use of so-called "Galois connections". They allow the introduction of new operators in such a way that a number of their properties is obtained simultaneously —for free in a manner of speaking. The chapter also contains a short introduction to the theory of Galois connections.

There are no new results in the chapter, but it contains the introduction (by a Galois connection) of a novel operator which is in essence the (weakest) liberal precondition. In later chapters this operator is used to formalise reductivity and termination.

Chapter 3 is organised around several different but equivalent formalisations of "is well-founded" and "admits induction". Although it is well known that well-founded and admitting induction are equivalent, proofs of that fact invariably entail the use of complementation. Our goal is to see what can be learnt by studying the two notions individually and with respect to each other in the context of a relation algebra in which complementation is not permitted. This is not only to play a nice formal game: in a later chapter the two notions are generalised. And the generalised notions are not equivalent anymore, not even when complementation is allowed.

For each of the two notions we first recall one of the usual pointwise definitions and then reformulate these definitions more concisely in the point-free style of relation algebra. To show the advantages of concision we prove some (well-known) facts such as that admits-induction implies well-foundedness (even in the absence of negation), and that the transitive closure of a relation is as well-founded as the relation itself.

With the understanding thus gained we proceed in Chapter 4 to tackle the construction of a number of proofs of less straightforward facts such as Newman's lemma [74], a lemma that is much exploited in the construction of term rewriting systems but is regarded as difficult to prove. We demonstrate that the proof of the lemma becomes straightforward by reducing it to purely syntactic considerations. As another example a sufficient condition for the well-foundedness of the union of two well-founded relations is calculated. The result generalises the known condition: "the union of two well-founded relations is well-founded if the two relations quasi-commute" [3]. This generalisation more or less comes out of the calculation, showing one advantage of the calculational method. It is even the case that we were not able to prove this result using the standard methods such as deriving a contradiction by constructing infinite chains. (Inevitably, such a proof was found by several of our colleagues when this thesis was almost complete.)

Chapter 5 is the first one to address programming. The chapter concentrates on how to implement relational specifications using repetitions. First the question of what the input-output relation of a repetition (*while* program) should be is answered. Then we proceed to express termination. An immediate consequence of our approach, as we demonstrate, is that it is not necessary to define the input-output relation of a repetition as a *least* solution (the standard approach) as long as attention is restricted to terminating programs. Any solution will do because, for a terminating repetition, there is only one solution. This seems to be a new result: it is not necessary to define the input-output behaviour of a terminating program as an extreme solution of an equation.

The, by now standard, way to construct repetitions is by using invariants, guards and bound functions. To relate this method of programming to the point-free relational approach the invariance theorem is formulated and proved.

In addition, some simple programming exercises are solved. This is done in such a way that it becomes clear that the method of using invariants is not necessarily restricted to imperative programming: it could also be exploited in the construction of functional (and possibly logic) programs.

In Chapter 6 the notions well-founded and admitting induction from Chapter 3 are generalised by introducing a datatype in the definitions. We prove that although "well-founded" and "admitting induction" are equivalent the obvious generalisations are not, the generalisation of "admitting induction" being the stronger of the two. We call the generalisation of admitting induction *reductivity*. The chapter also contains the result that reductivity is a generalisation of the categorical notion known as initiality and argues why, from a programmer's point of view, the generalisation is a useful one. This chapter shows an advantage of the point-free style: it is not easy to translate the point-free definitions to concise pointwise counterparts.

Reductivity is central in the termination proofs of recursive programs. It is therefore important to be able to prove that a relation has this property and also to construct relations enjoying the property. Therefore the chapter makes a start on developing a calculational theory of reductivity that is intended to guide the construction of such proofs and programs.

In Chapter 7 the theory of Chapter 5 is extended A broad class of recursive programs is defined and subsequently their input-output relations and termination condition are given. The class includes the structural recursive programs, also known as catamorphisms. However, it is not limited to these. For instance, it also contains the repetition and divide-and-conquer programs such as quicksort, which are certainly not structural recursive.

In Chapters 5 and Chapter 7 attention was restricted to terminating programs only. One of the insights that comes out of these chapters is that it is convenient to express the meaning of a recursive program using two relations. The first is the input-output relation, the second expresses the termination condition. The topic of Chapter 8 is a relational semantics of programs based on this insight. This is done in such a way that it, first, becomes possible to study also not necessarily terminating programs and, second, that the class of programs is extended to arbitrary recursive programs.

The main result of this chapter is that it is possible to avoid the so-called Egli-Milner order altogether. This order on programs is the standard vehicle in other approaches to relational semantics to define solutions of recursive program equations. Because it can be avoided we are able to present a simpler treatment and we also introduce a "fair choice" operator on programs without any difficulties. This operator resembles Mc Carthy's "amb" operator [66] and does give rise to problems in the more traditional approach.

The chapter gives the semantics of Dijkstra's guarded command language, the approach, however, can be used equally well to treat simple applicative languages. This provides evidence for the fact that when using a relational approach to semantics the differences between imperative and functional programming almost disappear.

# Chapter 2

# Basic Notions

## 2.1 Introduction

This chapter introduces the basic notions that are used throughout the text. It starts with a brief introduction to the theory of Galois connections. Then the basics of the relation calculus are presented. Although the axioms of this calculus are given, the treatment is rather informal. For instance, we do not supply proofs for all of the properties. Only a few proofs are provided and these are mainly meant to give an illustration of the techniques used in the rest of the text. The reason why we do not present all proofs is that it is not our intention to provide an axiomatic introduction to the relation calculus. We just want is to make the reader familiar with the rules of the calculus and with the pointfree formulation of some of the facts that will be used in later chapters. Once these rules and facts are known, they can (and will) be used in formal proofs. In this sense all of them can be considered as axioms.

In this chapter we do not discuss termination of programs: the only aspect of a program that is considered to be of interest is its input-output relation. To familiarise the reader with the calculus, we therefore often interpret relations as programs and, conversely, programs as relations. We sometimes even identify these and discuss relations in an operational way. It is hoped that in this way it becomes apparent that the rules of the relation calculus can be seen as useful laws of programming.

In the same informal vein, we present an introduction to our way of treating datatypes in the framework of the relation calculus. Specifically, the various ways of representing sets as relations are discussed. Then we introduce the cartesian product and disjoint sum of two sets. Also, it is explained how recursive datatypes like the natural numbers and the lists can be constructed in the relation calculus.

This chapter contains a discussion of the transitive closure of a relation, a concept that will be relevant for the relational approach to the semantics of the repetition. The main reason for including it is, however, to be able to give some typical examples of proofs in the relation calculus.

It is assumed that the reader is familiar with the predicate calculus and the very beginning of lattice theory. For instance, we assume the fixed point theorem of Knaster and Tarski [83] to be known. It will also help if the reader has some knowledge of imperative and functional programming.

## 2.2 Galois connections

The concept of Galois connection is well-known, see e.g. [23, 78, 33], but perhaps not as well known as it deserves to be. Given two preorders $\leq$ and $\sqsubseteq$, two functions, $\mathsf{f}$ and $\mathsf{g}$ say, form a *Galois connection* if the following formula holds for all $\mathsf{x}$ and $\mathsf{y}$.

$$\mathsf{f.x} \leq \mathsf{y} \quad \equiv \quad \mathsf{x} \sqsubseteq \mathsf{g.y} \quad . \tag{2.1}$$

Function $\mathsf{f}$ will be called the *lower adjoint* and function $\mathsf{g}$ the *upper adjoint*. (These names are chosen because (2.1) is a special case of the categorical notion of adjoint situation.) Galois connections are interesting because as soon as we recognise one we get a large number of properties of the adjoints for "free", in a manner of speaking. First of all, we have the two *cancellation properties*

$$\mathsf{x} \sqsubseteq \mathsf{g.(f.x)} \qquad \text{and} \qquad \mathsf{f.(g.y)} \leq \mathsf{y} \quad .$$

These are obtained by instantiating (2.1) in such a way that either the lefthand or the righthand side becomes true. Furthermore we have that both $\mathsf{f}$ and $\mathsf{g}$ are monotone. To demonstrate how nicely calculations with Galois connections proceed we show this for $\mathsf{f}$.

$$\begin{array}{ll}
 & \mathsf{f.x} \leq \mathsf{f.y} \\
\equiv & \qquad \{ \;\; \text{shunting: (2.1)} \;\; \} \\
 & \mathsf{x} \sqsubseteq \mathsf{g.(f.y)} \\
\Leftarrow & \qquad \{ \;\; \text{cancellation: } \mathsf{y} \sqsubseteq \mathsf{g.(f.y)}\text{; transitivity of } \sqsubseteq \;\; \} \\
 & \mathsf{x} \sqsubseteq \mathsf{y}
\end{array}$$

This calculation, albeit short, demonstrates the style of working with Galois connections: the use of the "shunting" rule (2.1) combined with the cancellation rule. Of course, in more substantial examples there is typically a number of additional steps after the shunting to make the cancellation possible.

If the two orders are complete lattices (which is, for instance, the case for the standard inclusion order on relations) something more can be said. Then the lower adjoint distributes over arbitrary joins and the upper adjoint distributes over arbitrary meets. As a consequence, the lower adjoint is bottom-strict and the upper adjoint top-strict.

Another fact, that is at least as important for our purposes, is that if a function on a complete lattice distributes over arbitrary joins then it has a unique upper adjoint and, conversely, if it distributes over arbitrary meets then it has a unique lower adjoint. This fact is of major importance because it allows us to introduce new operators: e.g. if we have a function $f$ on a complete lattice that distributes over arbitrary joins then we know that there is a unique function $f^\sharp$ with the shunting and cancellation properties:

$$f.x \sqsubseteq y \;\equiv\; x \sqsubseteq f^\sharp.y \quad \text{and} \quad f.(f^\sharp.x) \sqsubseteq x. \tag{2.2}$$

As an example of the introduction of such a function consider a complete, completely distributive lattice. Let $\sqcap$ denote the meet operator of the lattice and $y$ be an element of the lattice. That the lattice is completely distributive means that the function defined by $f.x = y \sqcap x$ distributes over arbitrary joins. The theory of Galois connections then allows to introduce the upper adjoint function $f^\sharp$ which in this case is denoted by $y\to$ . And we get immediately some properties:

$$y \sqcap x \sqsubseteq z \;\equiv\; x \sqsubseteq y \to z \quad \text{and} \quad y \sqcap (y \to x) \sqsubseteq x\;.$$

In the first property we recognise the familiar shunting property of the predicate calculus. The second (cancellation) corresponds to the rule of “modus ponens”. Readers with knowledge of Heyting algebras may appreciate this as a concise way of introducing them: a Heyting algebra is a lattice such that all “sections” $y\sqcap$ of the meet operator have an upper adjoint. (Given a binary operator $\oplus$ and an element $y$ the functions $(x \mapsto x \oplus y)$ and $(x \mapsto y \oplus x)$, also written shortly as $\oplus y$ and $y\oplus$, are called “sections” of $\oplus$ .)

In later sections of this chapter a number of operators on relations will be defined in the same way, i.e. as adjoints of more familiar ones. An advantage of this method is that it is not necessary to search for closed forms. (By a closed form we mean an expression in terms of the standard operators.) This is an advantage indeed because these closed forms tend to be rather complicated. Furthermore, if an operator is introduced as a closed form it is often not at all clear from the resulting formulae that the shunting and cancellation properties are valid. In summary: the recognition of the fact that an operator can be introduced via a Galois connection is often vital to effective calculation with the operator.

### 2.2.1 Galois connections and fixed points

In the previous section it was remarked that the typical pattern of a calculation with Galois connections consists of a combination of shunting and cancellation steps. In this section one such pattern is “canned” in a lemma. The reason is that the pattern occurs so often in calculations that it would be remiss not to do so.

The lemma concerns the relationship between Galois connections and fixed points. Suppose $h$ is an endofunction on some set partially ordered by the relation $\preceq$. A *fixed point* of $h$ is an element $x$ of the domain of $h$ such that $x = h.x$ . A *prefix point* of $h$ is an element $x$ of the domain of $h$ such that $h.x \preceq x$ . A logical question to ask is whether there are any general theorems relating the prefix points of two functions.

For the purposes of this thesis it suffices to restrict our attention to the consideration of complete lattices. In such a context we may apply the Knaster-Tarski fixed point theorem with which we assume familiarity. (We use the theorem in the following form: every monotone endofunction on a complete lattice has a least prefix point and a greatest postfix point, and both of these are fixed points of the function.)

So, let $(A, \leq)$ and $(B, \preceq)$ be complete lattices, and $g \in A \leftarrow A$ and $h \in B \leftarrow B$ be monotone endofunctions. We denote the least prefix point of $g$ by $\mu g$ and the least prefix point of $h$ by $\mu h$. The question we ask is: given function $f \in B \leftarrow A$ under what conditions can we establish a relationship between $\mu g$ and $\mu h$ in terms of $f$.

It is easy to derive a condition under which $\mu h \preceq f.\mu g$. Specifically:

$$
\begin{array}{ll}
 & \mu h \preceq f.\mu g \\
\Leftarrow & \quad \{ \text{ definition of } \mu h \ \} \\
 & h.(f.\mu g) \preceq f.\mu g \\
\Leftarrow & \quad \{ \text{ by definition of } \mu g,\ g.\mu g \leq \mu g \text{ assumption: } f \text{ is monotone } \} \\
 & h.(f.\mu g) \preceq f.(g.\mu g) \\
\Leftarrow & \quad \{ \ y := \mu g \ \} \\
 & \forall (y :: \ h.(f.y) \preceq f.(g.y)) \quad .
\end{array}
$$

We have thus derived that, for monotone $f$,

$$
\mu h \preceq f.\mu g \ \Leftarrow\ \forall (y :: \ h.(f.y) \preceq f.(g.y)) \quad . \tag{2.3}
$$

Property (2.3) is *not* by itself at all interesting: the weakening in the last step of its proof is very coarse. The property becomes interesting, however, when we combine it with the result of the investigation of the other inclusion, $f.\mu g \preceq \mu h$. The problem with this one is that the expression "$f.$ "prevents us from using Knaster-Tarski. To be able to shunt the expression to the other side of the inclusion we make the assumption that $f$ is the lower adjoint in a Galois connection. With $f^\sharp$ denoting the upper adjoint we calculate as follows:

$$
\begin{array}{ll}
 & f.\mu g \preceq \mu h \\
\equiv & \quad \{ \text{ shunting } \} \\
 & \mu g \leq f^\sharp.\mu h \\
\Leftarrow & \quad \{ \text{ (2.3) functions in a Galois connection are monotone } \} \\
 & \forall (y :: \ g.(f^\sharp.y) \leq f^\sharp.(h.y)) \\
\equiv & \quad \{ \text{ shunting } \} \\
 & \forall (y :: \ f.(g.(f^\sharp.y)) \preceq h.y) \quad .
\end{array}
$$

Were we now able to "move" the $f$ next to the $f^\sharp$ on the last line, cancellation (2.2) would be applicable. If, in addition, we can introduce an $h$ on the lefthand side that expression can be removed by monotonicity. Therefore, we assume $\forall (x :: \ f.(g.x) \preceq h.(f.x))$ and continue:

$$
\forall (y :: \ f.(g.(f^\sharp.y)) \preceq h.y)
$$

$$\begin{array}{ll} \Leftarrow & \{ \text{ assumption; transitivity } \} \\ & \forall(y :: h.(f.(f^\sharp.y)) \preceq h.y) \\ \Leftarrow & \{ \text{ monotonicity of } h \ \} \\ & \forall(y :: f.(f^\sharp.y) \preceq y) \\ \equiv & \{ \text{ cancellation: } y \preceq f.(f^\sharp.y) \ \} \\ & \text{true.} \end{array}$$

We thus conclude: if $f$ has an upper adjoint,

$$f.\mu g \preceq \mu h \Leftarrow \forall(x :: f.(g.x) \preceq h.(f.x)) \quad . \tag{2.4}$$

This lemma we call the *basic fusion lemma*.

A final step in this investigation is to find out when the inclusion in the left side of (2.4) can be strengthened to an equality. Combining (2.4) and (2.3) we obtain a theorem that is called the $\mu$-*fusion lemma*

**Lemma 2.5 ($\mu$-fusion)** If $g$ and $h$ are monotone functions on a complete lattice (with order $\preceq$) and if, furthermore, $f$ is a function that has an upper adjoint, then:

$$f\,.\,\mu g = \mu h \Leftarrow \forall(x :: f.(g.x) = h.(f.x))$$

(where $\mu g$ denotes the least fixed point of $g$).
□

(There is a stronger fusion lemma demanding only that $f$ be continuous and bottom strict rather than the lower adjoint in a Galois connection — the difference is finite distributivity — but proofs that we know of are substantially more complicated, and we know of no application where the additional strength is needed. More widely known is an incomparable theorem in which all three functions are required to be continuous.)

For a discussion of $\mu$-fusion and other fixed point lemmas see [65]. For a complete account of the theory of Galois connections, including proofs of the properties not proven here, see [1].

## 2.3 Relation calculus: the basic operators

### 2.3.1 The lattice structure

A binary relation on a set $\mathcal{U}$ is a set of pairs $(x,y)$ such that both $x$ and $y$ are elements of $\mathcal{U}$. In other words a relation is an element of the powerset $\mathcal{P}(\mathcal{U} \times \mathcal{U})$. Therefore, the first axiom of the relational calculus is that the relations form a complete, complemented lattice. The top of this lattice is denoted by $\top\!\!\top$, its interpretation being the total relation $\mathcal{U} \times \mathcal{U}$. The bottom, denoted by $\perp\!\!\perp$, has as interpretation the empty relation. The complement of relation $R$ is written $\neg R$.

By convention, variables $\mathsf{R}$, $\mathsf{S}$, $\mathsf{T}$, etc. will henceforth denote relations. Usually we write $x\langle\mathsf{R}\rangle y$ instead of $(x,y)\in\mathsf{R}$. It is assumed that when we interpret relations as programs, they have input from the right. This is unlike the convention in imperative programming, but in functional languages it is quite usual.

The relations $\top\!\!\top$ and $\perp\!\!\perp$ can be interpreted as programs as follows. The first is the program that can do anything: given an arbitrary input it may produce an arbitrary output. This program is called havoc. On the other hand the second program can do nothing: it does not accept any input and can not produce any output. It is often called magic or miracle.

We write $\subseteq$ for the order on the lattice. If relations are interpreted as programs (or as specifications), this order can be regarded as a refinement order. Program $\mathsf{R}$ refines program $\mathsf{S}$ means that any behaviour of $\mathsf{R}$ is also a behaviour of $\mathsf{S}$. In other words for any $x$ and $y$, if $x\langle\mathsf{R}\rangle y$ then also $x\langle\mathsf{S}\rangle y$. Or, expressed point free: $\mathsf{R}\subseteq\mathsf{S}$. A somewhat simplified view of programming is then: starting from a relation $\mathsf{S}$, the specification, find a program $\mathsf{R}$, the implementation, such that $\mathsf{R}$ refines $\mathsf{S}$.

For instance, any $\mathsf{R}$ refines $\top\!\!\top$: $\mathsf{R}\subseteq\top\!\!\top$. This expresses that any program is a refinement of havoc. And indeed refinement calculi have this rule, often expressing it as "all programs are as least as good as havoc". Another example is that magic is a refinement of any program: $\perp\!\!\perp\subseteq\mathsf{R}$. This is because $\perp\!\!\perp$ does not have any behaviour and therefore "can do nothing wrong". Of course this does not mean that, starting from a specification, the programming task consists simply of giving the program magic as an implementation. The reason is that magic is not considered to be an implementable program because it is not total. A total program is in this context a program that can accept all inputs.

For the union on relations the symbol $\cup$ is used. If $\mathsf{R}$ and $\mathsf{S}$ are interpreted as programs their union $\mathsf{R}\cup\mathsf{S}$ can be interpreted as a non-deterministic choice between these: $x\langle\mathsf{R}\cup\mathsf{S}\rangle y$ holds iff $x\langle\mathsf{R}\rangle y$ or $x\langle\mathsf{S}\rangle y$. Union is used in programming to obtain total implementations from non-total ones. For instance if $\mathsf{R}\subseteq\mathsf{S}$ and $\mathsf{T}\subseteq\mathsf{S}$ then also $\mathsf{R}\cup\mathsf{T}\subseteq\mathsf{S}$.

The intersection (meet) of two relations is denoted by $\cap$. Most programming languages do not have an operator which corresponds to intersection of programs. (The exceptions are languages for logic programming like Prolog.) However, it is an important one to structure specifications. For instance, suppose that $\mathsf{R}$ and $\mathsf{S}$ are relations on lists such that $x\langle\mathsf{R}\rangle y$ holds if $x$ is a permutation of $y$, and $x\langle\mathsf{S}\rangle y$ holds if $x$ is an ascending list. Then the intersection $\mathsf{R}\cap\mathsf{S}$ can be regarded as a specification of sorting.

Our precedence conventions are as follows. Unary operators always take precedence over binary operators. The precedence of the binary operators begins at the bottom end with the logical operators: first $\equiv$, then $\Leftarrow$ and $\Rightarrow$, followed by $\wedge$ and $\vee$. Then follow the lattice operators: inclusion $\subseteq$ has been given a precedence lower than $\cap$ and $\cup$ which have equal precedence. Other binary operators introduced later in the chapter have higher precedence than the lattice operators.

### 2.3.2 Composition

An operator on relations, to which we give a higher precedence than intersection and union, is composition. Relation composition is similar to program composition:

$$x\langle R{\circ}S\rangle y \;\equiv\; \exists(z : x\langle R\rangle z : z\langle S\rangle y)$$

That is, the composition of two relations relates two elements iff there exists an "intermediate" element. Relation composition is associative and has as unit the identity relation, so we have as an axiom $(I,\circ)$ is a monoid. The interpretation of $I$ as a relation is the identity relation: $x\langle I\rangle y \equiv x{=}y$. Its interpretation as a program is $\mathsf{skip}$.

An additional axiom is that the sections $(R\circ)$ and $(\circ R)$ distribute over arbitrary unions. A consequence of this axiom is that the relation $\perp\!\!\perp$ is a left and right zero of composition; furthermore, the operator $\circ$ is monotone in both its arguments with respect to $\subseteq$. From now on, if we say that an operator is monotone, it is to be understood that monotonicity is with respect to $\subseteq$.

As an example of a calculation with relations we now prove that $\top\!\!\top\circ\top\!\!\top = \top\!\!\top$ in a point free way.

$$
\begin{array}{ll}
 & \top\!\!\top \\
= & \qquad \{\ \ I \text{ is the identity of composition}\ \ \} \\
 & \top\!\!\top\circ I \\
\subseteq & \qquad \{\ \ I\subseteq\top\!\!\top\ ;\ \text{composition is monotone}\ \ \} \\
 & \top\!\!\top\circ\top\!\!\top \\
\subseteq & \qquad \{\ \ \top\!\!\top \text{ is the top}\ \ \} \\
 & \top\!\!\top
\end{array}
$$

### 2.3.3 Factors

An immediate consequence of the fact that both sections of composition distribute over arbitrary unions is the existence of the *factor* operators. These are defined by the Galois connections:

$$R{\circ}X\subseteq Y \;\equiv\; X\subseteq R\backslash Y \quad\text{and}\quad X{\circ}R\subseteq Y \;\equiv\; X\subseteq Y/R\,. \tag{2.6}$$

The most frequently used cancellation rules of the factors are:

$$R\circ R\backslash Y\subseteq Y \quad\text{and}\quad Y/R\circ R\subseteq Y\,. \tag{2.7}$$

These two operators have been given a variety of names in the literature, the oldest being the right and left *residual* operators [39]. We prefer Conway's [31] terminology viz. right and left *factor* operators. The cancellation rules show why: they are related to composition the same way as in arithmetic division is related to multiplication. For the same reason the operators are called *division* operators in [44].

Factor operators have been given a higher precedence than composition.

The factor operators have a number of properties that can easily be derived using the defining Galois connection. These properties are of course a consequence of related properties of composition. For instance from monotonicity of composition it follows that the function $(R\backslash)$ is monotone and the function $(\backslash Y)$ is anti-monotone. From the associativity of composition we infer that factor operators are "pseudo-associative":

$$R\backslash(T\backslash S) = (T\circ R)\backslash S \quad \text{and} \quad (R/T)/S = R/(S\circ T)\ . \tag{2.8}$$

Another consequence of the associativity of composition is that factor operators associate with each other:

$$R\backslash(T/S) = (R\backslash T)/S \quad . \tag{2.9}$$

The operators could have the following use in programming. Suppose we have to implement specification $Y$ and we have a program $R$ available such that, for a yet to be determined program $X$, $R\circ X\subseteq Y$. Then, because of (2.7), we can take $R\backslash Y$ as a specification for program $X$. This is why Hoare and He [53] call the factor $R\backslash Y$ a (weakest) *pre-specification*. Although the operators could be a nice tool for program construction, we will hardly use them. Instead another factor operator is used, which is introduced in Section 2.7.

### 2.3.4 Relation converse

It is easier to construct a program that flattens a parse tree than a program that solves the inverse problem, i.e. a parser. The process of "inverting" a given program is therefore considered to be "more than fun" in [29]. Relation calculus provides an operator $\cup$, the converse, to do exactly that. Formally: $x\langle R\cup\rangle y \equiv y\langle R\cup\rangle x$. The converse is connected to the lattice structure of the relations by the Galois connection

$$R\cup\subseteq X \quad \equiv \quad R\subseteq X\cup \quad . \tag{2.10}$$

(That the converse can be defined by a Galois connection was shown by Feijen and Van Gasteren in [41].) As a consequence, converse distributes over all lattice operators. In particular, it is monotone and distributes over empty unions and intersections: $\perp\!\!\perp\cup = \perp\!\!\perp$ and $\top\!\!\top\cup = \top\!\!\top$. From the two cancellation rules of Galois connection (2.10), $R\subseteq R\cup\cup$ and $X\cup\cup\subseteq X$, we obtain by antisymmetry that converse is its own inverse: $R\cup\cup = R$.

It is even the case that converse is a lattice isomorphism: $R\subseteq S \equiv R\cup\subseteq S\cup$. To prove this it suffices to start with the righthand side, apply shunting (2.10) once and then use that converse is its own inverse. Of course, we then also have by antisymmetry that $R{=}S \;\equiv\; R\cup{=}S\cup$. To obtain the axiom connecting converse and relation composition, we have to see that the converse of a composition of two relations is the "backward" composition of the converses of the two relations. Or, more precisely and concisely:

$$(R\circ S)\cup = S\cup\circ R\cup \quad . \tag{2.11}$$

From the interpretation it is clear that the converse of the identity relation is the identity relation. And it is also easy to prove this using (2.11):

$$
\begin{array}{ll}
 & \mathrm{I}^{\cup} \\
= & \{ \text{ I is the identity of composition } \} \\
 & \mathrm{I}^{\cup} \circ \mathrm{I} \\
= & \{ \text{ converse is its own inverse } \} \\
 & \mathrm{I}^{\cup} \circ \mathrm{I}^{\cup\cup} \\
= & \{ \text{ (2.11) } \} \\
 & (\mathrm{I}^{\cup} \circ \mathrm{I})^{\cup} \\
= & \{ \text{ I is the identity of composition } \} \\
 & \mathrm{I}^{\cup\cup} \\
= & \{ \text{ converse is its own inverse } \} \\
 & \mathrm{I}
\end{array}
$$

.

### 2.3.5 Other axioms

In the preceeding sections the calculus of relations is introduced by structuring it in three layers. First we introduced the lattice structure, then the composition monoid and, finally the converse structure. Furthermore, rules connecting the layers, such as the distribution of composition over union and (2.11) were given. This axiomatisation is not complete in the sense that not all properties of relations that are of interest to us can be derived. Some of these properties can be derived if two additional axioms are added.

The first of these is known as the *cone rule*. It is needed to prove that the top relation and the identity relation both differ from the empty relation. Therefore, it excludes the model formed by the relations on the empty universe.

The cone rule is formulated as follows:

$$\top\!\!\top \circ \mathsf{X} \circ \top\!\!\top = \top\!\!\top \quad \equiv \quad \mathsf{X} \neq \perp\!\!\perp \tag{2.12}$$

Because $\top\!\!\top \circ \top\!\!\top = \top\!\!\top$, the lefthand side of (2.12) reduces to *true* if $\mathsf{X}$ is instantiated to either $\top\!\!\top$ or I. Hence, these two relations indeed differ from the empty relation.

A second additional axiom connects all three layers of the calculus. The axiom is called the Dedekind rule [78] and is also known as the law of modularity [44]. It reads:

$$\mathsf{R} \circ \mathsf{S} \cap \mathsf{T} \quad \subseteq \quad (\mathsf{R} \cap \mathsf{T} \circ \mathsf{S}^{\cup}) \circ \mathsf{S} \quad . \tag{2.13}$$

This complicated axiom is almost never used explicitly in this thesis. Its importance lies in the earlier stages of the development of the relation calculus. For instance, some properties of the *monotypes*, to be introduced in the next section, can only be proved using this axiom. However, after these properties are established, one seldom, if ever, encounters the need to use the Dedekind rule.

## 2.4 The transitive closure

Inevitably our discussion of induction and well-foundedness will involve the notion of the reflexive transitive closure of a relation. For relation $\mathsf{R}$, we denote its reflexive and transitive closure by $\mathsf{R}^*$. It is usually defined as the least solution of the equation

$$\mathsf{X} :: \mathrm{I} \cup \mathsf{X} \circ \mathsf{R} \subseteq \mathsf{X} \quad . \tag{2.14}$$

The least solution exists, because (intersection and composition being monotone) the left-hand side of (2.14) is a monotone function of $\mathsf{X}$. Since relations form a complete lattice, it follows from the fixed point theorem of Knaster and Tarski that this function has a least fixed point. By definition $\mathsf{R}^*$ is a solution of (2.14), i.e. $\mathrm{I} \cup \mathsf{R}^* \circ \mathsf{R} \subseteq \mathsf{R}^*$, which is equivalent to the two properties $\mathrm{I} \subseteq \mathsf{R}^*$ ($\mathsf{R}^*$ is reflexive) and $\mathsf{R}^* \circ \mathsf{R} \subseteq \mathsf{R}^*$ .

In the same way, the reflexive closure of relation $\mathsf{R}$, denoted by $\mathsf{R}^+$, is defined as the least solution of the equation

$$\mathsf{X} :: \mathsf{R} \cup \mathsf{X} \circ \mathsf{R} \subseteq \mathsf{X} \quad . \tag{2.15}$$

Properties of these closure operators are, of course, well-known. How to establish them using relation algebra and, in particular, $\mu$-fusion and the existence of the factor operators is not. Just to show how nicely calculations in the algebra proceed we prove some of the properties of the closure operators.

As we noticed before, we have $\mathrm{I} \subseteq \mathsf{R}^*$ and $\mathsf{R}^* \circ \mathsf{R} \subseteq \mathsf{R}^*$. By monotonicity of composition it now immediately follows that $\mathrm{I} \circ \mathsf{R} \subseteq \mathsf{R}^*$, hence the reflexive, transitive closure of $\mathsf{R}$ contains $\mathsf{R}$. It is harder to prove that $\mathsf{R}^*$ is transitive, i.e. that $\mathsf{R}^* \circ \mathsf{R}^* \subseteq \mathsf{R}^*$, so we postpone a proof of this.

First, we generalise the two equations (2.14) and (2.15) to

$$\mathsf{X} :: \mathsf{S} \cup \mathsf{X} \circ \mathsf{R} \subseteq \mathsf{X} \quad . \tag{2.16}$$

Now we show that the least solution $\mathsf{Y}$ of (2.16) equals $\mathsf{S} \circ \mathsf{R}^*$:

$$\begin{array}{ll}
 & \mathsf{S} \circ \mathsf{R}^* = \mathsf{Y} \\
\Leftarrow & \qquad \{ \quad \mu\text{-fusion (2.5), } (\mathsf{S} \circ) \text{ has an upper adjoint; for any } \mathsf{X} \quad \} \\
 & \mathsf{S} \circ (\mathrm{I} \cup \mathsf{X} \circ \mathsf{R}) = \mathsf{S} \cup (\mathsf{S} \circ \mathsf{X}) \circ \mathsf{R} \\
\equiv & \qquad \{ \quad \text{calculus} \quad \} \\
 & \text{true} \quad .
\end{array}$$

By taking $\mathsf{R}$ for $\mathsf{S}$ we obtain that $\mathsf{R} \circ \mathsf{R}^* = \mathsf{R}^+$. But is also $\mathsf{R}^* \circ \mathsf{R} = \mathsf{R}^+$:

$$\begin{array}{ll}
 & \mathsf{R}^* \circ \mathsf{R} = \mathsf{R}^+ \\
\Leftarrow & \qquad \{ \quad \mu\text{-fusion (2.5), } (\circ \mathsf{R}) \text{ has an upper adjoint; for any } \mathsf{X} \quad \} \\
 & (\mathrm{I} \cup \mathsf{X} \circ \mathsf{R}) \circ \mathsf{R} = \mathsf{R} \cup (\mathsf{X} \circ \mathsf{R}) \circ \mathsf{R} \\
\equiv & \qquad \{ \quad \text{calculus} \quad \} \\
 & \text{true} \quad .
\end{array}$$

Both these calculations are very short, illustrating the utility of the fusion rule. However, sometimes the proof obligation that remains after applying the rule can not be fulfilled. An example of this situation arises when one attempts to prove transitivity of $R^*$, i.e. $R^*\circ R^* \subseteq R^*$, by means of $\mu$-fusion. However, transitivity of $R^*$ can be easily proved by observing that $R^*\circ R^*$ is the least solution of the equation:

$$X :: R^* \cup X\circ R \subseteq X \quad .$$

An appeal to the Knaster-Tarski fixed point theorem then suffices to prove that $R^*\circ R^* \subseteq R^*$.

It is a very educational exercise to rework the other well-known properties of the reflexive, transitive closure operator using $\mu$-fusion. This however is not the place for such an exercise, and from now on we will assume the validity of several properties without further ado. For instance, we assume known that $R^*$ is also the least solution of the equation in $X$: $I \cup R\circ X \subseteq X$ as well as the least solution of the equation in $X$: $I \cup X\circ X \cup R \subseteq X$ .

## 2.5 Sets as relations

Given a relation $R$, the *left domain* and *right domain* of $R$ are defined as follows. Its left domain is the set $\{x \mid \exists(y :: x\langle R\rangle y)\}$ . Its right domain is the set $\{x \mid \exists(y :: y\langle R\rangle x)\}$ . (These are commonly called the domain and range of the relation. We prefer "left" and "right" domain in order not to introduce an artificial and unnecessary direction to relations: relations are not always identified with programs!)

One of the advantages of relation algebra is that it is possible to represent sets as relations. Calculations with sets thus become special cases of calculations with relations. In particular, calculations with domains remain within the calculus itself and do not need to be conducted in some other formal framework. There are however several mechanisms for viewing sets as relations each of which has its own merits. Calculations are often made significantly more effective if one has a good grasp of exactly what the merits and demerits of each is.

One mechanism for representing sets as relations is via so-called "monotypes" (sometimes called "coreflexives" [44]), a second is via "left conditions" and a third via "right conditions" (sometimes called "row" and "column vectors" [80]). Axiomatically, these have the following definitions. First: we say that relation $A$ is a *monotype* iff $A \subseteq I$ . (By convention, variables $A$, $B$, $C$, etc. will henceforth denote monotypes.) Second: we say that relation $p$ is a *right condition* iff $p = \top\!\!\top\circ p$ . Third: we say that relation $p$ is a *left condition* iff $p = p\circ\top\!\!\top$ .

It is clear that for any given universe $\mathcal{U}$ there is a one-to-one correspondence between the subsets of $\mathcal{U}$ and the monotypes. Specifically, the set $a$ is represented by the monotype $A$ where $x\langle A\rangle y \equiv x = y \wedge x \in a$ . Equally clear is the existence of a one-to-one correspondence between the subsets of $\mathcal{A}$ and the right conditions on $\mathcal{A}$ . That is, if $a$ is some set then the right condition defined by $a$ is that relation $A_r$ such that for all $x$ and $y$, $x\langle A_r\rangle y \equiv y \in a$ . Similarly, the left condition corresponding to $a$ is that relation $A_l$ such that for all $x$ and $y$, $x\langle A_l\rangle y \equiv x \in a$ .

Using monotypes to represent subsets of $\mathcal{U}$ as relations, a restriction on a relation is modelled by composition of the relation with such a monotype, either on the left or on the right. Thus, if $R$ and $S$ are relations and $A$ is a monotype then $A\circ R$ and $S\circ A$ are both relations, the first being interpreted as the program $R$ after restricting its output to those elements in (the interpretation of) $A$, and the second being interpreted as the program $S$ after restricting its input to those elements in $A$. (Recall our convention that when relations are viewed as programs their input is on the right and their output on the left.) Note that in this interpretation composing a relation with a monotype is similar to adding an assertion or a guard to a program. For instance, the alternative construct $\mathsf{if}\ a \rightarrow r\ [\!]\ b \rightarrow s\ \mathsf{fi}$ can be interpreted as the relation $R\circ A \cup S\circ B$ , where $A$ and $B$ are the monotypes corresponding to guards $a$ and $b$, respectively, and $R$ and $S$ are the interpretation of $r$ and $s$ as relations.

Using conditions, a restriction on the left domain of relation $R$ is modelled by the intersection of $R$ with a left condition, and a restriction on the right domain of $R$ by its intersection with a right condition. The interpretation of the aforementioned alternative construct then looks something like $(R\cap ra) \cup (S\cap rb)$.

We now return to the discussion of the representation of the domains of a relation. The right domain of a relation $R$ is the smallest set such that restriction of the relation on the right to elements of that set has no effect on the relation. In other words: the right domain of a relation is the set of its possible input values. If we choose to represent sets by right conditions in relation algebra then it is quite obvious how to represent the right domain of $R$: it is right condition $\top\!\!\top\circ R$ since

$$\forall(p:\ p=\top\!\!\top\circ p:\ p\cap R = R \ \equiv\ \top\!\!\top\circ R\subseteq p)\ . \tag{2.17}$$

The straightforward proof of (2.17) makes use of only the lattice and monoid structures of a relation algebra. If, on the other hand, we choose to represent sets by monotypes then the right domain of $R$ is represented by the formula $I\cap \top\!\!\top\circ R$ . This is an uglier formula than $\top\!\!\top\circ R$ because it combines two operators (intersection and composition) whose behaviour relative to each other is complicated. Proof of the corresponding claim to (2.17),

$$\forall(A:\ A\subseteq I:\ R\circ A = R \ \equiv\ I\cap \top\!\!\top\circ R \subseteq A)\ , \tag{2.18}$$

demands the use of the Dedekind rule (2.13), and thus all three substructures in a non-complemented relation algebra.

These considerations seem to suggest that right conditions are the best way to represent right domains, and, dually, left conditions are the best way to represent left domains. The prominent role of composition in relation algebra (and programming), however, argues for the other choice. The point is that by choosing to represent sets by *monotypes* one can exploit to the full the calculational benefit of the associativity of composition. Thus, if $A$ is a monotype and $R$ and $S$ are relations, the composition $R\circ A\circ S$ can either be read as $(R\circ A)\circ S$ —a restriction on the right domain of $R$— or as $R\circ(A\circ S)$ —a restriction on the left domain of $S$. If one chooses to represent sets by left and/or right conditions then one must invent calculational rules that allow one to transform one type of restriction into the other.

The most effective calculations in relation algebra recognise and exploit the individual merits of conditions and monotypes, and thus involve a continual interplay between the two representations. This interplay can be seen in action in some of the calculations in these sections on sets as relations. Roughly speaking, general properties of domains are often easily *established* by using the condition representation but become most *useful* when expressed in terms of monotypes.

### 2.5.1 The isomorphism formalised

Representing sets by right conditions, the right domain of a relation is constructed by applying the function $(X \mapsto \top\!\top \circ X)$ . As we did before, this function is denoted briefly by $(\top\!\top\circ)$.

As remarked above, in the calculus of relations the right conditions are in one-to-one correspondence with the monotypes, i.e. there is a bijection to conditions from monotypes. This bijection is the function $(\top\!\top\circ)$ restricted to the monotypes. Since, however, the function $(\top\!\top\circ)$ is a total function on *all* relations, it is desirable to look for a total function on *all* relations that has as range the monotypes and is the inverse of $(\top\!\top\circ)$ when its domain is restricted to right conditions. There is a closed form for this function namely the function $(X \mapsto \top\!\top\circ X \cap I)$ . Since this closed form is unwieldy, it is preferable to avoid its use altogether. Instead we introduce the symbol " $>$ " written as a postfix to its argument to denote the function and define it to be the *lower* Galois adjoint of the function $(\top\!\top\circ)$ restricted to monotypes. That is, for all relations $R$ and all monotypes $A$,

$$R> \subseteq A \equiv R \subseteq \top\!\top\circ A \quad . \tag{2.19}$$

We call this operator the *right domain operator*.

The *left* domain operator is defined in a similar way. We have, for all relations $R$ and all monotypes $A$,

$$R< \subseteq A \equiv R \subseteq A\circ\top\!\top \quad . \tag{2.20}$$

The existence of the domain operators is guaranteed by the axioms stated in section 2.3, in particular the Dedekind rule (2.13). Since the calculations are not relevant to the current discussion we omit them here, and refer the reader to [8, 2]. We are now in a position to express formally in relation algebra the isomorphism between conditions and monotypes. Specifically, the right domain operator maps right conditions to monotypes, the function $(\top\!\top\circ)$ is its inverse: for all monotypes $A$ and all right conditions $p$

$$A = (\top\!\top\circ A)> \quad , \tag{2.21}$$

$$p = \top\!\top\circ p> \quad . \tag{2.22}$$

Yet more can be said. The function $(\top\!\top\circ)$ is in fact a lattice isomorphism between the lattice of monotypes and the lattice of right conditions. That is, for all monotypes $A$ and $B$,

$$A \subseteq B \equiv \top\!\top\circ A \subseteq \top\!\top\circ B \quad . \tag{2.23}$$

(The right domain operator, being inverse to $(\top\!\!\top\circ)$, is thus also such a lattice isomorphism.) Moreover, the rules (2.21) and (2.22) can both be made more general. Specifically, for all relations $R$,

$$(\top\!\!\top\circ R)> \; = \; R> \quad , \tag{2.24}$$

and

$$\top\!\!\top\circ R> \; = \; \top\!\!\top\circ R \quad . \tag{2.25}$$

Of course, for these rules there exist dual rules for left conditions and left domains. For convenience we list the properties below:

$$A = (A\circ\top\!\!\top)< \quad , \tag{2.26}$$

$$A\subseteq B \equiv A\circ\top\!\!\top \subseteq B\circ\top\!\!\top \quad , \tag{2.27}$$

$$R<\circ\top\!\!\top \; = \; R\circ\top\!\!\top \quad , \tag{2.28}$$

and

$$(R\circ\top\!\!\top)< \; = \; R< \quad . \tag{2.29}$$

### 2.5.2 Some other properties of monotypes and domains

In this section we collect a number of properties that are used in later chapters. Again, we do not include proofs of all of them, the interested reader is referred to [11].

Recall that a monotype is defined as a subset of the identity relation. This suggests that it should be easy to verify that taking the converse of a monotype does not have any effect, i.e. $A\cup = A$ for any monotype $A$. Although it is indeed possible to prove this using the relation calculus, it is not as simple as one might expect. The major reason for this is that the result depends on the (complicated) Dedekind rule. Another fact for which the same holds (it is easy to see in the standard model of the calculus, but has a rather complicated proof) is that for monotypes intersection and composition are the same: $A\circ B = A\cap B$. Because the two facts are so easy to be seen true in the standard model, we assume them without further ado. (For our purposes they may be considered as additional axioms.)

To illustrate the use of the domain operators, we now present some proofs of their properties. A first one is very simple: from the interpretation it is clear that the right domain of a relation is the left domain of the converse of that relation: $R> = (R\cup)<$. This is easy to prove using the defining Galois connections.

$$
\begin{array}{cl}
 & (R\cup)< \subseteq A \\
\equiv & \quad \{ \text{ domain shunting: } (2.20) \; \} \\
 & R\cup \subseteq A\circ\top\!\!\top \\
\equiv & \quad \{ \text{ converse shunting: } (2.10) \; \}
\end{array}
$$

$$
\begin{array}{ll}
 & R \subseteq (A \circ \top\!\!\top)^{\cup} \\
\equiv & \quad \{ \quad (2.11) \quad \} \\
 & R \subseteq \top\!\!\top^{\cup} \circ A^{\cup} \\
\equiv & \quad \{ \quad \top\!\!\top^{\cup} = \top\!\!\top \text{ ; converse is the identity on monotypes} \quad \} \\
 & R \subseteq \top\!\!\top \circ A \\
\equiv & \quad \{ \quad \text{domain shunting: (2.19)} \quad \} \\
 & R_{>} \subseteq A
\end{array}
$$

Hence, $(R^{\cup})_{<} \subseteq A \;\equiv\; R_{>} \subseteq A$. The desired equality now follows by instantiating $A$ to $(R^{\cup})_{<}$ this yielding $R_{>} \subseteq (R^{\cup})_{<}$. The other inclusion follows by instantiating $A$ to $R_{>}$ . This type of proof where an equality $X = Y$ is shown by proving that, for any $Z$, $X \subseteq Z \equiv Y \subseteq Z$ is known as a proof by *indirect equality*. The reason why this technique is useful in proving equalities involving operators defined via Galois connections is, that it enables one to use the shunting rules of the Galois connections. For this, the previous calculation is a typical example.

In the introduction of the domain operator it was remarked that the formula $\top\!\!\top \circ R \cap I$ is a closed form for the right domain of relation $R$. To give one example of the use of the Dedekind rule we now show that this is also true for the expression $I \cap R^{\cup} \circ R$ .

$$
R_{>} \;=\; I \cap R^{\cup} \circ R \tag{2.30}
$$

We prove $I \cap R^{\cup} \circ R \;=\; I \cap \top\!\!\top \circ R$. The inclusion $\subseteq$ follows from $R^{\cup} \subseteq \top\!\!\top$ and monotonicity of composition and intersection. For the other inclusion it suffices to show that $I \cap \top\!\!\top \circ R \;\subseteq\; R^{\cup} \circ R$ .

$$
\begin{array}{ll}
 & I \cap \top\!\!\top \circ R \\
\subseteq & \quad \{ \quad \text{Dedekind: } X \cap Y \circ Z \subseteq (X \circ Z^{\cup} \cap Y) \circ Z \quad \} \\
 & (I \circ R^{\cup} \cap \top\!\!\top) \circ R \\
= & \quad \{ \quad \text{calculus} \quad \} \\
 & R^{\cup} \circ R
\end{array}
$$

As a third property of domains we prove:

$$
(R \circ S_{<})_{<} = (R \circ S)_{<} \quad . \tag{2.31}
$$

Property (2.31) has a close connection to the rule $wlp.(r;s) = wlp.r \circ wlp.s$ from predicate transformer semantics. This will be discussed in Section 2.7. The proof is included because it illustrates our claim that it is often convenient to establish properties of the domain operators by using the condition representation.

We first exploit the fact that the function $(\circ\, \top\!\!\top)$ is a lattice isomorphism: instead of showing (2.31) we prove the equivalent $(R \circ S_{<})_{<} \circ \top\!\!\top \;=\; (R \circ S)_{<} \circ \top\!\!\top$ .

$$
\begin{array}{ll}
 & (\mathsf{R} \circ \mathsf{S}<) < \circ \top\!\!\top \\
= & \quad \{ \quad \text{domains: } \mathsf{R}< \circ \top\!\!\top = \mathsf{R} \circ \top\!\!\top \text{ , i.e. (2.28)} \quad \} \\
 & (\mathsf{R} \circ \mathsf{S}<) \circ \top\!\!\top \\
= & \quad \{ \quad \text{associativity of composition} \quad \} \\
 & \mathsf{R} \circ (\mathsf{S}< \circ \top\!\!\top) \\
= & \quad \{ \quad \text{again (2.28)} \quad \} \\
 & \mathsf{R} \circ (\mathsf{S} \circ \top\!\!\top) \\
= & \quad \{ \quad \text{associativity} \quad \} \\
 & (\mathsf{R} \circ \mathsf{S}) \circ \top\!\!\top \\
= & \quad \{ \quad \text{and a third time (2.28)} \quad \} \\
 & (\mathsf{R} \circ \mathsf{S}) < \circ \top\!\!\top
\end{array}
$$

In this calculation the use of associativity of composition was made explicit. We will not usually do so and use associativity tacitly.

## 2.6 Functionality and totality

A subset of the relations is formed by the functions, which can be seen as deterministic relations. There are a number of ways to characterise them and because we want to stress the importance of Galois connections we choose the following.

**Definition 2.32 (functions)** A relation $\mathsf{f}$ is called a function if and only if it has the property that for all relations $\mathsf{R}$ and $\mathsf{S}$:

$$\mathsf{R} \circ \mathsf{f}> \subseteq \mathsf{S} \circ \mathsf{f} \;\equiv\; \mathsf{R} \circ \mathsf{f}^{\cup} \subseteq \mathsf{S} \quad . \tag{2.33}$$

□

The definition of a function is almost a Galois connection, but not quite: the right domain on the lefthand side spoils it. However, it is a Galois connection if we restrict our attention to total functions, that is functions with right domain I. Another way of turning the definition into a Galois connection is by considering the set of relations with a right domain contained in $\mathsf{f}>$. It can be shown that these relations form a complete lattice with $\perp\!\!\perp$ as bottom element, relation $\top\!\!\top \circ \mathsf{f}>$ as top and the intersection and union operators as meet and join. It is not difficult to verify that the functions $(\circ \mathsf{f})$ and $(\circ \mathsf{f}^{\cup})$ form a Galois connection between this lattice and the lattice of the relations. As a consequence we then get that the function $(\circ \mathsf{f})$ distributes over non-empty intersections of relations, a property that is expected from pointwise considerations.

In this section we deviated from our practice of starting with a point-wise interpretation. So we now have to check whether the definition captures the idea of a function. The characterising property of a function is that it is single-valued, (also known as Leibniz' principle) i.e. if $x\langle f\rangle y$ and $z\langle f\rangle y$ then $x$ is equal to $z$. This is written as:

$$\forall(x,z:\ \exists(y::\ x\langle f\rangle y \wedge z\langle f\rangle y):\ x\langle I\rangle z)$$

After rewriting the existential quantification using relation composition and subsequently the universal quantification using the definition of relation inclusion, we obtain the (much more concise):

$$f \circ f^{\cup} \subseteq I \quad . \tag{2.34}$$

Expression (2.34) follows easily from definition 2.32 by instantiation of $R$ to the identity and $S$ to $f^{\cup}$. It is also not difficult to derive condition (2.33) from (2.34), in other words, expression (2.34) is an alternative definition of the notion of functional relation.

As is often the case with important concepts, there is a number of equivalent definitions of functionality. We mention a third one:

$$f \circ f^{\cup} = f^{<} \quad . \tag{2.35}$$

This one is obtained by rewriting (2.34) as $f \circ f^{\cup} = I \cap f \circ f^{\cup}$ and noting that the righthand side of the latter formula is equal to the left domain of $f$ (this uses the dual of (2.30).

The notion dual to functionality, viz. injectivity, is now of course easy to define as: $f$ is injective if and only if $f^{\cup}$ is a function. A relation that is both injective and functional is called a *bijection*.

Besides functionality and injectivity, there are two other dual notions which relations may enjoy: totality and surjectivity. We only spell out what it means for a relation to be total, because surjectivity can be defined in terms of totality: relation $R$ is surjective iff the converse $R^{\cup}$ is total.

Relation $R$ is total means that it can accept every element of the universe as an input, which is an alternative way of saying: $R^{>} = I$. An equivalent formulation is: $I \subseteq R^{\cup} \circ R$. From this it can be seen that surjectivity is in a sense also dual to injectivity: relation $R$ is injective can be expressed as $I \supseteq R^{\cup} \circ R$.

## 2.7 The monotype factor

In predicate transformer semantics there exists a well-known connection between the weakest precondition function and Hoare triples:

$$A \Rightarrow \mathrm{wlp}.R.B \quad \equiv \quad \{A\}R\{B\} \quad . \tag{2.36}$$

Because the two concepts have proven to be important for program construction we want to add them to our calculus. The interpretation of $\{A\}R\{B\}$ is, first, that program $R$ terminates, when restricted to inputs satisfying condition $A$. In the present discussion it is assumed that all programs are guaranteed to terminate on every input. (Here programs and relations are identified and for relations the property "non-terminating" does not really make sense. The reader may rest assured: termination will be discussed in later chapters.) The second fact expressed by $\{A\}R\{B\}$ is that, in reaction to inputs satisfying $A$, program $R$ produces only outputs satisfying $B$. To express this in relation calculus, we remark: relation $R$ restricted to inputs from monotype $A$ is the relation $R{\circ}A$ (recall that input is considered to be from the right). The set of possible outputs of a relation is its right domain, thus the fact that relation $S$ "produces" outputs in the set $B$ can be expressed as: $S{<} \subseteq B$.

Now the righthand side of (2.36) can be expressed by: $(R \circ A){<} \subseteq B$. This expression, viewed as a function in $A$, is the composition of the left domain operator and the function $(R\circ)$. Both these functions are the lower adjoint in a Galois connection and, therefore, distribute over arbitrary unions. Of course, this has as a consequence that their composition also distributes over arbitrary unions, and therefore has an upper adjoint. This adjoint then is the relational form of the weakest liberal precondition. We call this function $(R\backslash)$ the *monotype factor* of $R$, its defining Galois connection being:

$$A \subseteq R\backslash B \quad \equiv \quad (R{\circ}A){<} \subseteq B \quad . \tag{2.37}$$

So, the connection between the weakest precondition function and Hoare triples turns out to be a Galois connection. It should be noted that this is a Galois connection in the (complete) lattice of monotypes. Therefore, $R\backslash B$ is by definition a monotype.

The interpretation of $R\backslash B$ is the following:

$$x \in R\backslash B \equiv \forall(y : y\langle R\rangle x : y{\in}B) \quad . \tag{2.38}$$

Viewing relation $R$ as a program which takes its input from the right we see that $R\backslash B$ is the greatest set such that if program $R$ is started with a value from that set as input, the result will be in the set $B$: $R\backslash B$ is, indeed, a weakest (liberal) precondition. Note, furthermore, that $x \in R\backslash B$ holds vacuously when $x \notin R{>}$, that is when $x$ is not a possible input of $R$.

By duality, there exists a second factor operator defined by the connection

$$A \subseteq B/R \equiv (A{\circ}R){>} \subseteq B \quad . \tag{2.39}$$

The properties of that operator are of course completely dual to the one defined in (2.37) and therefore it will not be discussed any further.

Because the monotype factor plays a prominent role in this thesis we give, for referential purposes, a rather long list of properties. All of these are easy to prove using the defining Galois connection and the properties of the domain operator documented in [2]. Also, we give some connections with the properties of the weakest precondition function to support the interpretation.

First note that the function $(R\backslash)$ , being an upper adjoint, distributes over arbitrary intersections of monotypes. This corresponds to the fact that the function wlp.$r$ is universally conjunctive. In particular, function $(R\backslash)$ is top-strict: $R\backslash I = I$ (of course in this case $I$, the greatest monotype, is the top of the lattice). Because intersection on monotypes coincides with composition it follows that $(R\backslash)$ distributes over the operator $\circ$ .

Sometimes (see e.g. [37]) it is assumed that, for all $r$, the function wlp.$r$ is a bottom-strict function. (This is, in fact, assumed for the function wp.$r$. However, if we do not care about termination, the functions wlp.$r$ and wp.$r$ coincide.) This property of the weakest precondition function is known as "the law of the excluded miracle". This law is not generally valid for relations. We have:

$$R\searrow\perp\!\!\!\perp \;=\; \sim\!(R_>) \qquad , \tag{2.40}$$

where operator $\sim$ is the complement operator of the monotype lattice (i.e. $\sim\!A = \neg A \cap \mathrm{I}$). Because there is a Galois connection involved, the proof of the property is by indirect equality. For all monotypes $A$: .

$$
\begin{array}{cl}
& A \subseteq R\searrow\perp\!\!\!\perp \\
\equiv & \quad \{ \;\; (2.37) \text{ i.e. shunting } \;\; \} \\
& (R\circ A)_< \subseteq \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; \text{domains: } X_< = \perp\!\!\!\perp \;\equiv\; X_> = \perp\!\!\!\perp, \\
& \quad\;\;\;\; \text{both sides are equivalent to } X = \perp\!\!\!\perp \;. \;\; \} \\
& (R\circ A)_> \subseteq \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; \text{domains: the dual of } (2.31) \;\; \} \\
& (R_>\circ A)_> \subseteq \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; R_>\circ A \text{ is a monotype;} \\
& \quad\;\;\;\; \text{the domain operator is the identity on monotypes.} \;\; \} \\
& R_>\circ A \;\subseteq\; \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; \text{shunting in the monotype lattice} \;\; \} \\
& A \subseteq \sim\!(R_>)
\end{array}
$$

It follows from (2.40) that function $(R\searrow)$ is bottom-strict iff the relation $R$ is total. So, the law of the excluded miracle states that all programs are total, i.e. can accept all inputs. Therefore, the law does *not* hold for relations.

From (2.37) we obtain the cancellation property:

$$(R\circ R\searrow B)_< \subseteq B \qquad . \tag{2.41}$$

(The monotype factor has higher precedence than composition.) Sometimes, the cancellation rule is used in the following form, which is obtained by rewriting (2.41) using the general property $X_< \subseteq B \;\equiv\; X = B\circ X$ of the domain operator:

$$R\circ R\searrow B = B\circ R\circ R\searrow B \qquad . \tag{2.42}$$

Because $R\searrow B \subseteq \mathrm{I}$, (2.42) has as a consequence:

$$R\circ R\searrow B \subseteq B\circ R \qquad . \tag{2.43}$$

Both (2.41) and (2.43) express that program $R$ produces a result from set $B$ when started in a state satisfying $R\backslash B$. As will be shown, (2.43) can be rewritten as an equality if $R$ is a monotype:

$$A \circ A\backslash B \;=\; B{\circ}A \tag{2.44}$$

The calculation that proves this reads:

$$\begin{array}{ll}
 & B{\circ}A \;\subseteq\; A \circ A\backslash B \\
\equiv & \quad \{ \text{ on monotypes, composition is intersection } \} \\
 & B{\circ}A \;\subseteq\; A\backslash B \\
\equiv & \quad \{ \text{ shunting: (2.37) } \} \\
 & (A{\circ}B{\circ}A){<} \;\subseteq\; B \\
\equiv & \quad \{ \text{ on monotypes, composition is intersection and} \\
 & \qquad {<} \text{ is the identity } \} \\
 & \text{true}
\end{array}$$

We call (2.44) the rule of modus ponens. The reason is that if monotypes are interpreted as predicates then, when restricted to monotypes only, the operator $\circ$ and the operator $\backslash$ are interpreted as conjunction and implication, respectively. Thus (2.44) is the statement $A \wedge A{\Rightarrow}B \;\equiv\; A{\wedge}B$.

Another consequence of the cancellation property that is sometimes useful is:

$$R{>} \circ R\backslash A \subseteq (A{\circ}R){>} \quad . \tag{2.45}$$

The righthand side of (2.45) is the set of possible inputs of the relation $A{\circ}R$: elements $x$ such that $R$ can accept $x$ as input (or $x \in R{>}$) and that when $R$ is started in $x$ there exists an output satisfying $A$. The lefthand side is the set of elements $x$ such that $x \in R{>}$ and that, when $R$ is started in $x$, the result is *guaranteed* to satisfy $A$. With this interpretation it comes as no surprise that when $R$ is a functional relation the inclusion in (2.45) can be strengthened to an equality:

$$R \text{ is a function} \qquad \Rightarrow \qquad R{>} \circ R\backslash A = (A{\circ}R){>} \quad . \tag{2.46}$$

This expresses that for functions (or deterministic programs), what may happen is the same as what is guaranteed to happen.

The weakest precondition function has the property $\mathrm{wlp}.r.(\mathrm{wlp}.s.a) = \mathrm{wlp}.(r;s).a$ . Translating this to a fact for the monotype factor we obtain:

$$R\backslash(S\backslash A) = (S{\circ}R)\backslash A \quad . \tag{2.47}$$

As an illustration we give a proof for this one. There is a Galois connection involved, therefore the proof uses indirect equality. For all monotypes $B$:

$$B \subseteq R\backslash(S\backslash A)$$
$$\equiv \qquad \{ \text{ factor shunting: (2.37) } \}$$
$$(R\circ B)_< \subseteq S\backslash A$$
$$\equiv \qquad \{ \text{ factor shunting: (2.37) } \}$$
$$(S \circ (R\circ B)_<)_< \subseteq A$$
$$\equiv \qquad \{ \text{ domains: (2.31) } \}$$
$$(S\circ R\circ B)_< \subseteq A$$
$$\equiv \qquad \{ \text{ factor shunting: (2.37) } \}$$
$$B \subseteq (S\circ R)\backslash A$$

Combining (2.44) and (2.47) we get the following:

$$A \circ R\backslash B = A \circ (R\circ A)\backslash B \quad . \tag{2.48}$$

Translated to preconditions it reads $(a \wedge wlp.r.b) = (a \wedge wlp.(?a\,;r).b)$ . Here, program $?a$ is a *guard*, a program with property $wlp.?a.b = (a \Rightarrow b)$. Guards are used, for instance in [49], to restrict the possible inputs of a program. Program $?a\,;r$ then corresponds to relation $R\circ A$.

A well-known fact concerning the weakest precondition function is that it distributes over nondeterministic choice, which for programs is denoted by $[\!]$ . The property reads for programs $wlp.(r\,[\!]\,s).a = (wlp.r.a \wedge wlp.s.a)$ . For relations this translates to

$$(R\cup S)\backslash A \quad = \quad R\backslash A \;\cap\; S\backslash A \quad , \tag{2.49}$$

because for relations nondeterministic choice is union. The property is easy to prove by indirect equality.

$$B \subseteq (R\cup S)\backslash A$$
$$\equiv \qquad \{ \text{ shunting: (2.37) } \}$$
$$((R\cup S)\circ B)_< \subseteq A$$
$$\equiv \qquad \{ \text{ composition distributes over union } \}$$
$$(R\circ B \cup S\circ B)_< \subseteq A$$
$$\equiv \qquad \{ \text{ the domain operator is a lower adjoint, therefore it distributes over union } \}$$
$$(R\circ B)_< \cup (S\circ B)_< \subseteq A$$
$$\equiv \qquad \{ \text{ defining property of union } \}$$
$$(R\circ B)_< \subseteq A \quad \wedge \quad (S\circ B)_< \subseteq A$$
$$\equiv \qquad \{ \text{ shunting: (2.37) } \}$$

$$
\begin{array}{ll}
 & B \subseteq R\backslash A \quad \wedge \quad B \subseteq S\backslash A \\
\equiv & \qquad \{ \text{ defining property of intersection } \} \\
 & B \subseteq R\backslash A \cap S\backslash A
\end{array}
$$

Finally we mention the monotonicity properties of the monotype factor; we have: $(R\backslash)$ is monotone and $(\backslash A)$ is antimonotone. This concludes our short, and rather informal, introduction to the monotype factor. For a more extensive and formal treatment the reader is referred to [12].

## 2.8 Relators and datatypes

In computing science there are two different understandings of what a datatype is. The first is: a datatype is a set together with a number of operations on it. This view is taken in the study of abstract datatypes. The standard example of such a datatype is of course the stack. Here the set, usually called the carrier of the datatype, is formed by all stacks. The operations of this datatype are push, pop, etc. Other examples are for instance priority queues and the booleans with operations like "and" and "or".

A second interpretation of a datatype is: it is just a set. This is the one used in typed programming languages like Pascal or Haskell: a variable is of type integer means that it is an element of the set of integers. Similarly, a variable is of type "$A \rightarrow B$" says that it is an element of the set of functions from set $A$ to set $B$. For the moment the second view is adopted here: a datatype is just a set. It turns out, however, that if a new datatype is constructed, usually a number of operations on the datatype emerge from the construction as well.

In programming there are various ways to construct datatypes. Usually a number of datatypes such as the integers and the booleans are assumed known. It is then possible to construct new datatypes as solutions of recursive equations. Examples of these are the natural numbers and the lists. New datatypes can also be formed from old by datatype constructors, either with or without the use of recursion. Well-known examples of constructions without recursion are the product or pairing operation and the disjoint union or sum of two sets: given two sets $A$ and $B$, their product $A \times B$ is the set of pairs $(x,y)$ such that $x$ and $y$ are elements of $A$ and $B$, respectively.

The disjoint union $A+B$ of the sets consists of elements of the form $\mathsf{tag0}.x$ and $\mathsf{tag1}.y$, where, again, $x$ is an element of $A$ and $y$ is taken from $B$. The tagging operations are such that it is decidable whether an element $z$ from $A+B$ has the form $\mathsf{tag0}.x$ or $\mathsf{tag1}.y$. Furthermore, it must be possible to strip the tag from its argument, in other words the tagging operations have an inverse. Therefore it is possible to distinguish between the cases "element $z$ represents an element of $A$" and "$z$ represents an element of $B$". So $A+B$ is indeed the *disjoint* union of $A$ and $B$.

A typical example of a datatype former that is defined by recursion is the list former. Given a set $A$, the set $\mathsf{list}.A$ consists of all finite lists over $A$. The list operation is defined by recursion, as is seen in the usual definition in functional programming languages: a list is either a constant (namely the empty list), or an element from $A$ "consed" with a list (where the "cons" operation is the well-known one from the programming language Lisp). So lists are defined in terms of lists, i.e. by a recursive definition. For each datatype former $F$ there exists a relation former Fr. Informally, Fr.$R$ is the relation "apply $R$ to each element". As a typical example consider lists; given functional relation $f$, the relation Listr.$f$ is the well-known function map.f: (Listr.$f$).$[x,y,z] = [f.x\,,f.y\,,f.z]$. It is not necessary to restrict Fr to functional relations: for arbitrary relation $R$ the relation Fr.$R$ relates lists of the same length, such that the elements "with the same index" are related by $R$. We, for instance, have: $[x,y,z]\langle \mathrm{Fr}.R\rangle[p,q,r] \equiv x\langle R\rangle p \wedge y\langle R\rangle q \wedge z\langle R\rangle r$.

In general: every datatype constructor $F$ is a function from datatypes to datatypes. Function $F$ maps a set $A$ to the set of what are called $F$-structures over $A$. Furthermore associated with $F$ is a relation constructor $Fr$ which maps a relation $R$ on a set $A$ to a relation on $F$-structures. Relation Fr.$R$ is to be thought of as "$R$ applied element-wise". In order to introduce datatype formers and the corresponding relation formers, we need two functions: a function that maps monotypes to monotypes —the type constructor— and a function from relations to relations. Because monotypes are a special case of relations, however, it turns out that these two functions can be combined into one: we only need to consider functions from relations to relations. Of course, not every function of this type will do, because not every such function is the combination of a datatype former with its associated relation former. Functions that are will be called *relators*. As for the properties that relators should have, we know that $F.R$ should be thought of as $R$ applied element-wise. This means that relators are monotone with respect to the standard order on relations. It also means that they distribute over relation composition and commute with the converse operator. (It is instructive to verify these properties for Listr.)

Of course we still want a relator to map sets to sets; this is the case if we require that $F.I \subseteq I$: given the monotonicity of $F$ this implies ( even equivales) that $F$ maps monotypes to monotypes. (It is necessary to check that in this respect it is harmless to identify sets with partial identity relations, in other words that if a monotype $A$ is viewed as a set, monotype $F.A$ is indeed the set of $F$-structures over $A$, whereas if $A$ is viewed as a relation, relation $F.A$ is "relation $A$ applied element-wise". Again, checking this for List is instructive.)

**Definition 2.50 (Relators)** A function $F$ from relations to relations is a relator if it enjoys the following properties.

**(a)** $F$ is monotone

**(b)** $F$ distributes over $\circ$

**(c)** $F$ distributes over $^{\cup}$ , i.e. over the converse operator

**(d)** $F.I \subseteq I$

□

Examples of relators are the identity relator $\mathsf{Id}$ ($\mathsf{Id}.R = R$) and the constant relators, i.e. the relators $F$ such that $F.R = A$ for a fixed monotype $A$. For a binary relator, that is for a function from pairs of relations to relations, the composition and converse operators are to be applied component-wise. Thus, for instance, the distribution of binary relator $\oplus$ over composition takes the shape

$$(R{\circ}U)\oplus(S{\circ}V) \quad = \quad (R{\oplus}S)\circ(U{\oplus}V) \quad .$$

### 2.8.1 Datatypes defined without recursion

The basic type constructors we consider are the constant relators, product, and disjoint sum. We shall not introduce them axiomatically, but merely present their relational interpretations and their main properties. ( From these properties it can be proved that the constructors are, indeed, relators. For a more extensive treatment see [9, 8].)

For each monotype $A$ , the constant function mapping arbitrary relation $R$ to $A$, is a relator. Besides the existence of the monotypes $\mathrm{I}$ and $\perp\!\!\!\perp$ we assume that there is a unit (mono)type, denoted $\mathbb{1}$. The unit type is a set with one (anonymous) element. It is called a unit type because it is a unit of the cartesian product operator: $\mathbb{1}{\times}A$ and $A$ are isomorphic. From the unit type we obtain the constant relator $(X{\mapsto}\mathbb{1})$.

It is furthermore assumed that the universe is closed both under summing and under product. ( Note that given the unit type and the sum, we may conclude that the universe is infinite.)

For monotypes $A$ and $B$, product $A{\times}B$ has the familiar interpretation: the set of pairs from $A$ and $B$. The left and right projection functions are denoted $\ll$ and $\gg$, respectively. Sum $A{+}B$ is interpreted as the disjoint union of $A$ and $B$ as it was introduced informally in the introduction of this section. The left and right tagging ( or, injection ) functions $\mathsf{tag0}$ and $\mathsf{tag1}$ are denoted $\hookrightarrow$ and $\hookleftarrow$, respectively. So, $\hookrightarrow$ and $\hookleftarrow$ are two total, injective functions with disjoint left domains.

For relations $R$ and $S$, the product $R{\times}S$ and the disjoint sum $R{+}S$ have, in a way, dual properties. In the following we only present relevant ones. We deal with product first. (To parse the formulae one has to know that all operators introduced in this section have been given a precedence higher than composition and lower than the factors.)

As mentioned earlier, the product and the sum of two relations can be viewed as an "apply to each element". This will become clear from the interpretations. We introduce the product first, together with its *generator* $\vartriangle$ (called *split*) , the essence of which is that it doubles arguments and thus generates pairs. Their interpretations are as follows.

$$(x,y)\langle R{\vartriangle}S\rangle z \quad \equiv \quad x\langle R\rangle z \wedge y\langle S\rangle z$$

$$(x,y)\langle R{\times}S\rangle(v,z) \;\equiv\; x\langle R\rangle v \wedge y\langle S\rangle z \quad .$$

(Note the link between product and parallel composition. A discussion of parallel composition in a relational framework can be found in [77].) Besides the general properties enjoyed by (binary) relators, a property used quite frequently is the following fusion rule (so called because it is a rule about "fusing" a composition of two relations into one relation):

**[$\times$-$\vartriangle$-fusion]** $R{\times}S \circ T{\vartriangle}U \;=\; (R\circ T)\vartriangle(S\circ U)$

Note that a special instance of this rule is $T{\vartriangle}U = T{\times}U \circ I{\vartriangle}I$, a property justifying our earlier remark that the essence of the split is its doubling arguments: the interpretation shows that $I{\vartriangle}I$ *is* the doubling function.

Unlike $I{\vartriangle}I$ , however, $R{\vartriangle}R$ when "applied" to an argument $x$, does not in general yield a pair of equal values ( it does if $R$ is a function): by the nondeterminacy of relations, first applying $R$ and then doubling is not the same as first doubling the argument and then applying $R$ to each ; we only have the *inclusion* $I{\vartriangle}I \circ R \subseteq R{\vartriangle}R$ . As a consequence, in general composition does not distribute over $\vartriangle$ from the left; we just have an inclusion.

Another property with an inclusion where one might at first expect an equality is:

$$\ll \circ R{\vartriangle}S \subseteq R \quad\text{ and }\quad \gg \circ R{\vartriangle}S \subseteq S \;. \tag{2.51}$$

This time the reason is not the non-determinacy but the partiality: if, for instance, $S$ is equal to the empty relation $R{\vartriangle}S$ is also empty. Therefore, in that case, the lefthand side of the first formula in (2.51) is empty too. Thus if $R$ is non-empty we do not have an equality. There is, however, a rule similar to (2.51) with an equality: $\ll \circ R{\vartriangle}S = R \circ S{>}$. For the same reason we have for product:

$$\ll \circ R{\times}S \;=\; R \circ \ll \circ I \times S{>} \quad . \tag{2.52}$$

Of course, there is also a dual rule for the right projection.

Analogously to the product, the disjoint sum comes with a generator $\blacktriangledown$ (co-junc), which in essence boils down to a test, and a co-generator, the conditional denoted $\triangledown$ (junc). Operators junc and co-junc are each other's $\cup$-conjugate, i.e. $R{\blacktriangledown}S = (R^{\cup} \triangledown S^{\cup})^{\cup}$; as a result, properties of junc and co-junc come in pairs. Informally, the interpretation of operator $R{\triangledown}S$ is: first test how the input is tagged, untag the input, then apply either $R$ or $S$. So:

$$x\langle R{\triangledown}S\rangle{\hookrightarrow}.y \;\equiv\; x\langle R\rangle y \qquad\text{and } x\langle R{\triangledown}S\rangle{\hookleftarrow}.y \;\equiv\; x\langle S\rangle y$$

Formally, $\triangledown$ is defined as:

$$R{\triangledown}S \;=\; R \circ {\hookrightarrow}^{\cup} \cup S \circ {\hookleftarrow}^{\cup} \quad .$$

Disjoint sum $R{+}S$ is to be interpreted as "apply to each element of the input". In this case an input is of one of the two shapes ${\hookrightarrow}.y$ and ${\hookleftarrow}.y$; such an input contains the single element $y$. Therefore, applying to each element is: first untag, then apply the proper argument of the sum, then tag again. Formally, using $\triangledown$, this is rendered by

$$R{+}S \;=\; ({\hookrightarrow} \circ R) \triangledown ({\hookleftarrow} \circ S) \quad .$$

While product is related to parallel execution, disjoint sum is related to selection. To show this we shall exploit both the junc and the co-junc and, again, introduce some fusion rules. Note that $R{\blacktriangledown}S$ is equal to ${\hookrightarrow} \circ R \cup {\hookleftarrow} \circ S$; in particular, for monotypes $A$ and $B$ we have $A{\blacktriangledown}B = {\hookrightarrow} \circ A \cup {\hookleftarrow} \circ B$.Thus, $A{\blacktriangledown}B$ can be interpreted as a test on $A$ versus $B$, together with a tagging operation to distinguish the outcome of the test (which may be nondeterministic).

The fusion rules to be introduced are

| | | | |
|---|---|---|---|
| **[$+$-$\blacktriangledown$-fusion]** | $U{+}V \circ R{\blacktriangledown}S$ | $=$ | $(U \circ R) \blacktriangledown (V \circ S)$ |
| **[$\triangledown$-$+$-fusion]** | $R{\triangledown}S \circ U{+}V$ | $=$ | $(R \circ U) \triangledown (S \circ V)$ |
| **[$\triangledown$-$\blacktriangledown$-fusion]** | $R{\triangledown}S \circ U{\blacktriangledown}V$ | $=$ | $R \circ U \cup S \circ V$ |
| **[$\blacktriangledown$-$\circ$-fusion]** | $R{\blacktriangledown}S \circ T$ | $=$ | $(R \circ T) \blacktriangledown (S \circ T)$ |
| **[$\circ$-$\triangledown$-fusion]** | $T \circ R{\triangledown}S$ | $=$ | $(T \circ R) \triangledown (T \circ S)$ |

Note, again, the special instance of $+$-$\blacktriangledown$-fusion $R{\blacktriangledown}S \;=\; R{+}S \circ I{\blacktriangledown}I$ , a property justifying our earlier remark that, in essence, the co-junc boils down to a selection: The relation $I{\blacktriangledown}I$ is a nondeterministic selection of either a left or a right tag. Depending on which is chosen, $R{+}S$ subsequently executes either $R$ or $S$. Note also that, in contrast with the split, composition does distribute over junc (from the right, that is).

Now we are ready to explore the connection of all the above with selection. We consider the selection statement

$$\mathsf{if}\ A \to R \ [\!]\ B \to S\ \mathsf{fi} \quad ,$$

where $A$ and $B$ are guards ( monotypes), and $R$ and $S$ are statements ( relations). One way of rendering this selection relationally is, of course, as the union $R{\circ}A \cup S{\circ}B$ of $R$ restricted to $A$ and $S$ restricted to $B$. By $\triangledown$-$\blacktriangledown$-fusion, this is equal to

$$R{\triangledown}S \circ A{\blacktriangledown}B \quad , \tag{2.53}$$

which expresses more explicitly that an if-statement is a test followed by a statement. Applying some fusion rules to the latter formula, we find it to be equal to

$$I{\triangledown}I \circ R{+}S \circ A{+}B \circ I{\blacktriangledown}I \quad . \tag{2.54}$$

The correspondence between the selection statement and (2.54) is now obvious. Relation $I \blacktriangledown I$ corresponds to the "if", $A{+}B$ to the guards, $R{+}S$ to the branches and $I{\triangledown}I$ to the "fi". And thus, we hope, we have clarified the connection between disjoint sum and selection. There is one more connection to be mentioned: we consider (2.53) and compose it on the right side with relation $T$; distributing the latter over the junc yields the familiar

$$\mathsf{if}\ A \to R \ [\!]\ B \to S\ \mathsf{fi}\ ;T \quad = \quad \mathsf{if}\ A \to R;T \ [\!]\ B \to S;T\ \mathsf{fi} \quad .$$

After this discussion of the connection between sum and selection we conclude this section by giving the "cancellation" rules, comparable to (2.51) and (2.52), of junc and sum. This time they are what one expects from interpretation, i.e. with equality rather than just an inclusion. We formulate the rules that correspond to (2.51) in terms of the co-junc operator, because this is the one that is mostly used.

$$\hookrightarrow_{\cup} \circ R{\blacktriangledown}S = R \quad \text{and} \quad \hookleftarrow_{\cup} \circ R{\blacktriangledown}S = S \tag{2.55}$$

The dual rule for the junc operator is easily obtained by taking the converse. Finally, the rules for disjoint sum, comparable to (2.52) are:

$$\hookrightarrow_{\cup} \circ R{+}S = R \circ \hookrightarrow_{\cup} \quad \text{and} \quad \hookleftarrow_{\cup} \circ R{+}S = S \circ \hookleftarrow_{\cup} \tag{2.56}$$

### 2.8.2 Recursive datatypes and initial algebras

As an example of how datatypes are defined by recursion we consider the natural numbers. A natural number is either the constant zero or the successor of a natural number. The constant zero can be modelled as a functional relation, call it $\mathsf{zero}$, to the naturals from the unit type : $\mathsf{zero} \in \mathbb{N} \leftarrow \mathbb{1}$. The successor function $\mathsf{succ}$ has type $\mathbb{N} \leftarrow \mathbb{N}$. These two functions are known as the *constructors* of the natural numbers.

Every natural number is the result of applying a constructor, this is called the "no-junk property [1]" of the natural numbers with respect to the constructors. The no-junk property of the natural numbers can also be formulated as: the union of the left domains of the constructors is the natural numbers.

Given a natural number, there has to be a test to find out by which constructor it was formed and from which argument. In terms of relation algebra this means that the constructors have disjoint left domains and are injective. In datatype theory terms this is called: the "no-confusion property" of the natural numbers and its constructors.

Now consider the type of the constructors: they have the same left domain. This has as a consequence that they can be combined into one function of type $\mathbb{N} \leftarrow \mathbb{1}+\mathbb{N}$ using the junc operator: $\mathsf{zero} \triangledown \mathsf{succ}$. This function is an example of an *algebra*. In general, if we have a number of relations with the same left domain $\mathsf{A}$, they can be combined into one relation with left domain $\mathsf{A}$ by means of the junc operator. This will result in a relation with right domain contained in $\mathsf{F.A}$ for some relator $\mathsf{F}$. In the example the relator is $(\mathbb{1}+)$. It is common practice to call any relation $\mathsf{R}$ with the property that it has a left domain contained in some set $\mathsf{A}$ and a right domain contained in $\mathsf{F.A}$ an *algebra*, even if it is not formed by juncing a number of operations together. The type $\mathsf{A}$ is called the *carrier* of the algebra.

Next we return to the definition of the naturals. The requirements we have thus far, no junk and no confusion, can be reformulated now as: algebra $\mathsf{zero} \triangledown \mathsf{succ}$ is a bijection between $\mathbb{N}$ and $\mathbb{1}+\mathbb{N}$. This can be shown in some simple calculations. Of course, this is not enough to define the naturals: there are many types $\mathsf{A}$ and functions $\mathsf{f}$ such that $\mathsf{f}$ witnesses the fact $\mathsf{A} \cong \mathbb{1}+\mathsf{A}$. But there is one more requirement: induction over the natural numbers is valid. The requirement boils down to $\mathbb{N}$ being in a certain sense the least type $\mathsf{A}$ such that $\mathsf{A} \cong \mathbb{1}+\mathsf{A}$.

A problem of the form: find a bijection $\mathsf{A} \cong \mathsf{F.A}$ is called "solving a domain equation". The solution of such an equation consists of an algebra; the solution for $\mathsf{A}$ can then be recovered as the carrier of this algebra. Of course often a special kind of solution is required: the "least" one, which is called the *initial algebra*. This is in general a tricky problem: first we have to find out what least means in this context and, second, it is not at all clear for which class of relators such equations do have solutions.

Working in a relation algebra has the advantage that domain equations always have a "least solution": an initial $\mathsf{F}$-algebra exists for all relators $\mathsf{F}$. To see this we strengthen the equation $\mathsf{F.A} \cong \mathsf{A}$ to be an equality rather than just an isomorphism. Because relators are

[1] The use of the word "junk" here should not be confused with the name "junc" given to the co-generator of disjoint sum. The latter is short for "junction" and the former means waste or trash.

monotone functions from monotypes to monotypes and the monotypes form a complete lattice it is clear that there is a smallest monotype $A$ such that $A = F.A$, called $\mu F$. Then $\mu F$ is certainly an isomorphism between $F\,.\,\mu F$ and $\mu F$. So this gives the feature: carrier, algebra and its inverse are the same. Of course there is the need to check that our interpretation of "least" is reasonable; this question will be addressed in a later chapter.

It might not be clear how the constructors can be recovered from $\mu F$. As an example, we show how it is done for the naturals:

$$
\begin{array}{ll}
 & \mu(\mathbb{1}+) \\
= & \quad \{ \ \mu(\mathbb{1}+) \text{ is a fixed point of } (\mathbb{1}+) \ \} \\
 & \mathbb{1} + \mu(\mathbb{1}+) \\
= & \quad \{ \ \text{definition of } + \ \} \\
 & (\hookrightarrow \circ \mathbb{1}) \triangledown (\hookleftarrow \circ \mu(\mathbb{1}+))
\end{array}
$$

The constructor $\mathsf{zero}$ is $\hookrightarrow \circ \mathbb{1}$: the function that maps the element $*$ of the unit type to $\hookrightarrow.*$. This then is the number 0. The constructor $\mathsf{succ}$ is $\hookleftarrow \circ \mu(\mathbb{1}+)$: the function that maps a natural number $n$ to $\hookleftarrow.n$. So a natural number $n$ has the form $\hookleftarrow^n.(\hookrightarrow.*)$.

The type of Lists over a given set $B$ is defined in the same way as the natural numbers, this time taking domain equation $\mathsf{list}.B \cong \mathbb{1}+(A \times \mathsf{list}.B)$ as a starting point. The construction yields the two constructors $\mathsf{nil}$ and $\mathsf{cons}$, that can be derived in a similar way to what was done for the naturals. Note that for recursive datatypes we obtain besides a set also a number of operations and, thus, something which resembles a more classical datatype.

The construction of the natural numbers does not lead to an interesting datatype transformer: it only yields a datatype. For datatypes like List we do, however, get something interesting. The defining domain equation contains a monotype as a parameter. Therefore, the construction gives a datatype transformer: the function that maps a monotype $B$ into the monotype of lists over $B$. The question now remains whether this datatype transformer can be extended to a relator. This is indeed possible and how to do it will be discussed in chapter 7. Here we just briefly hint at what the construction looks like: with $\otimes$ denoting a binary relator, it can be shown that the function that maps $R$ to the least fixed point $\mu(R\otimes)$ is a relator too.

### 2.8.3 Natural transformations for monotypes

There are programs that have as inputs $F$-structures and output $G$-structures (for $F$ and $G$ relators) with the property that they do not introduce new elements in the output structures. Examples are the reverse function on lists, the projection functions, and the doubling function. The first, from List-structures to List-structures, does not affect the individual elements of its input list at all. The second, from "product structures" to "$\mathsf{Id}$-structures" throws elements of its input away. And, finally, the third makes a copy of the element in the input $\mathsf{Id}$-structure. Such relations are of importance because they preserve all properties of the elements of the structures they relate. If the input is an $F$-structure such that all its elements satisfy property $A$, the output is a $G$ structure for which the same holds. This is expressed point-free in the next definition.

**Definition 2.57** Let $R$ be a relation and let $F$ and $G$ be relators. We say that $R \in F \stackrel{\cdot}{\leadsto} G$ ($R$ is from $F$ to $G$) if and only if for all monotypes $A$:

$$R \circ F.A \subseteq G.A \circ R \quad .$$

□

When the property $R \in F \stackrel{\cdot}{\leadsto} G$ is expressed with points it becomes: if $y \langle R \rangle x$ and $x \in F.A$ then it follows that $y \in G.A$ . The fact that $x \in F.A$ is then interpreted as: all elements of the $F$-structure $x$ are in the set $A$ (or: satisfy property $A$). That $R$ does not introduce any new elements follows by instantiating $A$ to the set of all elements of the input structure $y$.

The property $R \in F \stackrel{\cdot}{\leadsto} G$ resembles the notion "$R$ is a natural transformation from $F$ to $G$", although the latter is stronger: then the definition must not only hold for all monotypes, but for arbitrary relations. This is why we call a relation from $F$ to $G$ a *natural transformation for monotypes*.

A, not very surprising, observation is that $R \in F \stackrel{\cdot}{\leadsto} G$ can also be expressed using the monotype factor.

**Lemma 2.58** $R \in F \stackrel{\cdot}{\leadsto} G \equiv \forall(A : A \subseteq I : F.A \subseteq R \searrow G.A)$

**Proof**

$$
\begin{array}{ll}
 & F.A \subseteq R \searrow G.A \\
\equiv & \quad \{ \text{ factors: } (2.37) \ \} \\
 & (R \circ F.A)< \subseteq G.A \\
\equiv & \quad \{ \text{ domains: } X< \subseteq A \ \equiv \ X \subseteq A \circ X \ \} \\
 & R \circ F.A \subseteq G.A \circ R \circ F.A \\
\equiv & \quad \{ \ (\Rightarrow)\text{: } F.A \subseteq I;\ (\Leftarrow)\text{: } F.A \circ F.A = F.A \ \} \\
 & R \circ F.A \subseteq G.A \circ R
\end{array}
$$

□

## 2.8.4 Some properties of relators

From the defining properties of a relator a great number of facts can be proved. Some of them are very easy such as: relators preserve functionality and totality. Others are not so straightforward. Two of these are that relators commute with the domain operators: $F.(R<) = (F.R)<$. Again, the difficulty of the proof is that it needs the use of the Dedekind rule. For a proof see [2].

Because sum and product are relators we have $(R+S)< = R<+S<$ and $(R \times S)< = R< \times S<$, but what about the domains of junc and split? From the interpretation of junc we expect

$(R\triangledown S)_<$ to be equal to $R_< \cup S_<$ and also $(R\triangledown S)_> = R_> + S_>$ . These two facts can indeed be proved in the relation calculus.

It is not possible to find an easy expression for the left domain of a split. But that is to be expected from the interpretation: the domain is not simply a product. It is something called a *dependent product*. However for the right domain something can be said: the right domain of a split is the intersection of the right domains of the operands: $(R\vartriangle S)_> = R_> \cap S_>$

The correspondence between disjoint sum and nondeterministic choice can, again, be seen in the following identity.

$$(R\blacktriangledown S)\backslash\!\!\cdot(A+B) = R\backslash\!\!\cdot A \cap S\backslash\!\!\cdot B \tag{2.59}$$

The proof is not difficult:

$$
\begin{array}{ll}
 & C \subseteq (R\blacktriangledown S)\backslash\!\!\cdot(A+B) \\
\equiv & \quad \{ \ \text{factor shunting: (2.37)} \ \} \\
 & (R\blacktriangledown S \circ C)_< \subseteq A+B \\
\equiv & \quad \{ \ \blacktriangledown\text{-}\circ\text{-fusion} \ \} \\
 & ((R\circ C)\blacktriangledown(S\circ C))_< \subseteq A+B \\
\equiv & \quad \{ \ (R\blacktriangledown S)_< = R_< + S_< \ \} \\
 & (R\circ C)_< + (S\circ C)_< \ \subseteq \ A+B \\
\equiv & \quad \{ \ \text{property of sum: it is injective} \ \} \\
 & (R\circ C)_< \subseteq A \ \wedge \ (S\circ C)_< \subseteq B \\
\equiv & \quad \{ \ \text{factor shunting: (2.37)} \ \} \\
 & C \subseteq R\backslash\!\!\cdot A \ \wedge \ C \subseteq S\backslash\!\!\cdot B \\
\equiv & \quad \{ \ \text{defining property of intersection} \ \} \\
 & C \subseteq R\backslash\!\!\cdot A \cap S\backslash\!\!\cdot B
\end{array}
$$

Finally we mention a property connecting relators and the monotype factor. Relators distribute over the monotype factor in the sense that

$$F.(R\backslash\!\!\cdot A) \subseteq F.R\backslash\!\!\cdot F.A \quad . \tag{2.60}$$

This property follows because relators distribute over composition and commute with the domain operators.

# Chapter 3

# Mathematical Induction Made Calculational

In computing science the notions of well-foundedness and admitting induction are of great importance. The first is used, for instance, in the proof of termination of programs. The second is used to prove properties of inductively defined data structures. This chapter argues the use of point-free relation algebra to formulate these fundamental notions. By doing so one obtains much compacter formulae than the pointwise formulae with which we are all familiar. As a result it is easier to understand the relationship between the two notions and to recognise the circumstances in which they are applicable. A second advantage is that it is easier to see how they can be generalised in a useful way. Finally, the invention of inductive hypotheses can sometimes be reduced to purely syntactic considerations. This, of course, can be of great help in the construction of calculational proofs.

This last fact in particular and the compactness of the formulae are of importance to us. What we ultimately want to have is a simple, calculational theory of induction and recursion. This theory should also be useful in practice. That is, it should be useful in the construction of inductive arguments and recursive programs. The present chapter is organised around several different but equivalent formalisations of "is well-founded" and "admits induction". For each of the two notions we first recall the usual pointwise definition and then reformulate their definitions more concisely in the point-free style of relation algebra. Then we proceed to formulate a yet more concise definition which we show equivalent to the original formulation.

It is known that "is well-founded" and "admits induction" are equivalent notions but proofs of that fact invariably entail the use of complementation. Our goal here is to see what can be learnt by studying the two notions individually and with respect to each other in the context of a relation algebra in which complementation is not permitted. This is *not* primarily because we want to play an entertaining formal game. As it will turn out in a later chapter, both notions can be generalised. However, we can not prove that the two generalised notions are equivalent, not even in the presence of negation, the generalised notion of well-foundedness being the weaker one. That means that we have to be somewhat careful with properties of well-founded relations. Generalisations of some of them might also hold for the generalised notion of admitting-induction and some might not. Studying the two notions in a non-complemented relation algebra can give an indication which properties generalise nicely and which don't.

In this chapter we also begin to develop a calculational theory of well-foundedness by treating some standard ways of constructing well-founded relations, such as the lexical coupling.

## 3.1 Well-foundedness

We now formulate well-foundedness in a point-free relation algebra. We do this in three ways which we then prove to be equivalent.

Expressed in terms of points, a relation $R$ is said to be well-founded if there are no infinite $R$-chains, i.e. there do not exist points $x_0, x_1, x_2, \ldots$ such that $x_{i+1}\, R\, x_i$ for all $i$, $i \geq 0$. To obtain a point-free formulation, we make the following observation. Let such an infinite chain be represented by a set. This set is characterised by the property that for an arbitrary member $x$ there exists a member $y$ with $y\langle R\rangle x$. Conversely, if we have such a set, an infinite $R$-chain can be constructed. Indeed, the elements $x$ of such a set have the property that an infinite chain can be constructed with $x$ as the starting point.

So, the fact that relation $R$ is well-founded can be expressed by the statement that all such sets are empty. (Note that for an empty set it is vacuously true that for any of its elements there is a member $y$ such that $y\langle R\rangle x$.) We arrive thus at: relation $R$ is well-founded iff for all sets $A$ it is the case that:

$$A = \phi \quad \Leftarrow \quad \forall(x : x{\in}A : \exists(y : y{\in}A : y\langle R\rangle x)) \quad .$$

Now we choose to represent sets by monotypes. Then, noting that $\exists(y : y{\in}A : y\langle R\rangle x)$ is the statement that $x$ is in the right domain of $A{\circ}R$ and the antecedent thus can be expressed as $A \subseteq (A{\circ}R)_>$, this definition converts directly into the following point-free form.

**Definition 3.1 (Monotype-well-founded)** Relation $R$ is said to be *monotype-well-founded* if and only if it satisfies

$$\forall(A :\; A \subseteq I :\; A \subseteq \perp\!\!\perp \Leftarrow A \subseteq (A{\circ}R)_>) \quad . \tag{3.2}$$

□

Characteristic of (3.2) is that it is a rule for establishing when a set represented by a monotype $A$ is empty. This shows why well-foundedness is such a nice property: it allows one to replace the statement that a set is empty with the formally much weaker statement $A \subseteq (A{\circ}R)_>$ .

Of course it is also possible to represent sets by conditions. Then the definition becomes:

**Definition 3.3 (Condition-well-founded)** Relation $R$ is said to be *condition-well-founded* if and only if it satisfies

$$\forall(p :\; p = \top\!\!\top{\circ}p :\; p \subseteq \perp\!\!\perp \Leftarrow p \subseteq p{\circ}R) \quad . \tag{3.4}$$

□

This definition is used as a stepping stone to the third definition in which sets are replaced by arbitrary relations.

**Definition 3.5 (Relation-well-founded)** Relation $\mathsf{R}$ is said to be *relation-well-founded* if and only if it satisfies

$$\forall(\mathsf{S} :: \ \mathsf{S} \subseteq \perp\!\!\perp \ \Leftarrow \ \mathsf{S} \subseteq \mathsf{S} \circ \mathsf{R}) \ \ . \tag{3.6}$$

□

This definition appears elsewhere under different names: for example "definite" [10], "progressively finite" [80] and "left-founded" [38]. Because of its elegantly compact form it has no doubt been introduced in many other publications.

The claim is that all three definitions of well-foundedness are equivalent. As is to be expected the equivalence between monotype- and condition-well-foundedness is a straightforward consequence of the isomorphism between monotypes and conditions.

**Theorem 3.7** For all relations $\mathsf{R}$, $\mathsf{R}$ is monotype-well-founded equivales $\mathsf{R}$ is condition-well-founded.

**Proof** With dummies $\mathsf{p}$ and $\mathsf{A}$ ranging over right conditions and monotypes, respectively, we have:

$$\begin{array}{ll}
 & \forall(\mathsf{p} :: \ \mathsf{p} \subseteq \perp\!\!\perp \ \Leftarrow \ \mathsf{p} \subseteq \mathsf{p} \circ \mathsf{R}) \\
\equiv & \qquad \{ \ \text{range translation: } \mathsf{A} \mapsto \top\!\!\top \circ \mathsf{A} \text{ is a bijection} \ \} \\
 & \forall(\mathsf{A} :: \ \top\!\!\top \circ \mathsf{A} \subseteq \perp\!\!\perp \ \Leftarrow \ \top\!\!\top \circ \mathsf{A} \subseteq \top\!\!\top \circ \mathsf{A} \circ \mathsf{R}) \\
\equiv & \qquad \{ \ \text{monotype-condition iso: (2.25) and} \\
 & \qquad \ \ \text{(2.23)} \ \} \\
 & \forall(\mathsf{A} :: \ \mathsf{A} \subseteq \perp\!\!\perp \ \Leftarrow \ \mathsf{A} \subseteq (\mathsf{A} \circ \mathsf{R})>) \ \ .
\end{array}$$

□

The equivalence between condition- and relation-well-founded takes only a little more work.

**Theorem 3.8** For all relations $\mathsf{R}$, $\mathsf{R}$ is condition-well-founded equivales $\mathsf{R}$ is relation-well-founded.

**Proof** Condition-well-foundedness of $\mathsf{R}$ is obviously implied by relation-well-foundedness of $\mathsf{R}$ (since every condition is also a relation). To prove the opposite implication assume that $\mathsf{R}$ is condition-well-founded. Then, for all $\mathsf{S}$ we have:

$$\begin{array}{ll}
 & \mathsf{S} \subseteq \perp\!\!\perp \\
\Leftarrow & \qquad \{ \ I \subseteq \top\!\!\top, \text{ hence by monotonicity of composition } I \circ \mathsf{S} \subseteq \top\!\!\top \circ \mathsf{S} \ \} \\
 & \top\!\!\top \circ \mathsf{S} \subseteq \perp\!\!\perp
\end{array}$$

$\Leftarrow$ { $\top\!\!\top \circ S$ is a condition assumption:
R is condition-well-founded: (3.4) }

$\top\!\!\top \circ S \subseteq (\top\!\!\top \circ S) \circ R$

$\equiv$ { associativity of composition }

$\top\!\!\top \circ S \subseteq \top\!\!\top \circ (S \circ R)$

$\Leftarrow$ { monotonicity of composition }

$S \subseteq S \circ R$

□

In view of theorem 3.8 we no longer make the distinction between "monotype", "condition" or "relation" well-founded; we say that R is *well-founded* if it satisfies any one of the three definitions.

We can give a yet more concise definition of well-foundedness. Observe that the fact $S \subseteq S \circ R$ can also be expressed as: relation S is a postfix point of the function $X \mapsto X \circ R$. ("Postfix point" is the notion dual to prefix point.) Therefore, definition (3.5) can be read as: any postfix point of the function $X \mapsto X \circ R$ equals the empty relation. In particular this holds for the greatest postfix point. But if this greatest postfix point is empty then it is of course the case that all postfix points are empty. So definition (3.5) is equivalent to: the greatest postfix point (which is by also, by Knaster-Tarski, the greatest fixed point) of the function $X \mapsto X \circ R$ is the empty relation.

**Definition 3.9 (Well-founded)** Relation R is well-founded is equivalent to $\nu(\circ R) = \perp\!\!\perp$.
□

As should be clear from the discussion using points that lead to the formulation of definition 3.1, the right condition $\nu(\circ R)$ represents the set of points which are the starting point of an infinite R-chain. Interpreted this way, definition (3.9) is exactly what one expects: there do not exist infinite R-chains.

Now a first lemma on well-foundedness is immediate. By monotonicity of the greatest fixed point operator it follows that if R is well-founded then also any relation contained in R has that property.

**Lemma 3.10** If relation R is well-founded and $S \subseteq R$ then S is well-founded.
□

## 3.2 Well-foundedness of the transitive closure

A well-known result is that R is well-founded if and only if its transitive closure $R^+$ is well-founded. We include a proof in order to illustrate the advantage of using a definition like definition 3.9.

**Theorem 3.11** For all $\mathsf{R}$, that $\mathsf{R}$ is well-founded equivales that $\mathsf{R}^+$ is well-founded.

**Proof** From lemma (3.10) it follows that if $\mathsf{R}^+$ is well-founded then $\mathsf{R}$ is well-founded, since $\mathsf{R} \subseteq \mathsf{R}^+$. It remains to prove the converse. We prove the more general statement that if $\mathsf{R}$ is well-founded and $\mathsf{T}$ is related to $\mathsf{R}$ by the property

$$\mathsf{T} \circ \mathsf{T} \subseteq \mathsf{T} \circ \mathsf{R} \tag{3.12}$$

then $\nu(\circ\mathsf{T})$ is contained in $\nu(\circ\mathsf{R})$. The required result follows because $\mathsf{R}^+$ is indeed so related to $\mathsf{R}$. We have:

$$\begin{array}{cl}
 & \nu(\circ\mathsf{T}) \subseteq \nu(\circ\mathsf{R}) \\
\Leftarrow & \qquad \{ \quad \text{Knaster-Tarski} \quad \} \\
 & \nu(\circ\mathsf{T}) \subseteq \nu(\circ\mathsf{T}) \circ \mathsf{R} \\
\equiv & \qquad \{ \quad \nu(\circ\mathsf{T}) = \nu(\circ\mathsf{T}) \circ \mathsf{T} \quad \} \\
 & \nu(\circ\mathsf{T}) \circ \mathsf{T} \circ \mathsf{T} \subseteq \nu(\circ\mathsf{T}) \circ \mathsf{T} \circ \mathsf{R} \\
\Leftarrow & \qquad \{ \quad \text{monotonicity and transitivity} \quad \} \\
 & \mathsf{T} \circ \mathsf{T} \subseteq \mathsf{T} \circ \mathsf{R}
\end{array}$$

□

This concludes this section. We have now established the equivalence of the five properties

- $\mathsf{R}$ is condition-well-founded
- $\mathsf{R}$ is monotype-well-founded
- $\mathsf{R}$ is relation-well-founded
- $\nu(\circ\mathsf{R}) = \perp\!\!\perp$
- $\mathsf{R}^+$ is (condition-, monotype- or relation-) well-founded.

## 3.3 The induction principle

A relation $\mathsf{R}$ is said to *admit induction* if the following schema can be used to establish that property $\mathsf{P}$ holds everywhere: prove, for all $y$, that the induction hypothesis $\forall(x : x\langle\mathsf{R}\rangle y : \mathsf{P}.x)$ implies $\mathsf{P}.y$ . That is, expressed in terms of points, $\mathsf{R}$ admits induction iff for all $\mathsf{P}$:

$$\forall(y :: \ \mathsf{P}.y) \Leftarrow \forall(y :: \ \forall(x : x\langle\mathsf{R}\rangle y : \mathsf{P}.x) \Rightarrow \mathsf{P}.y) \quad .$$

A well-known example of a relation that admits induction is the standard "less than" relation on the natural numbers. If this relation is substituted for $\mathsf{R}$ we obtain the induction principle known as *strong induction*. In this section we formulate the notion of admitting induction in relation algebra in a number of ways and then show the equivalence of these formulations.

The pointwise definition of "admits induction" given above is in terms of predicates. Because we want to arrive at a definition in terms of relations we first reformulate it in terms of sets. So we get: relation $R$ admits induction if and only if for all sets $A$:

$$\forall(y ::\ y{\in}A) \ \Leftarrow\ \forall(y ::\ \forall(x : x\langle R\rangle y : x{\in}A) \Rightarrow y{\in}A) \quad . \tag{3.13}$$

To obtain a definition without dummies we first notice that $\forall(y ::\ y{\in}A)$ , the (understood) domain of $y$ being I, can be rewritten as $I \subseteq A$ . Furthermore, we see that the expression in the domain of the antecedent, $\forall(x : x\langle R\rangle y : x{\in}A)$ , is just $y \in R\mathbin{\backslash\!\!\!\cdot}A$, see (2.38). So (3.13) can be drastically simplified to

$$I \subseteq A \Leftarrow R\mathbin{\backslash\!\!\!\cdot}A \subseteq A \quad , \tag{3.14}$$

for all monotypes $A$. To aid the intuition a bit: $R\mathbin{\backslash\!\!\!\cdot}A$ corresponds to what is usually called the induction hypothesis, while a proof of $R\mathbin{\backslash\!\!\!\cdot}A \subseteq A$ is a proof of the induction step.

This then is the first definition of "admits induction".

**Definition 3.15 (Monotype Induction)** The relation $R$ is said to *admit induction on monotypes* if and only if it satisfies

$$\forall(A : A \subseteq I : I \subseteq A \Leftarrow R\mathbin{\backslash\!\!\!\cdot}A \subseteq A) \quad . \tag{3.16}$$

□

If instead of representing sets by monotypes we choose to represent them by left conditions we arrive at the following definition:

**Definition 3.17 (Condition Induction)** The relation $R$ is said to *admit induction on (left) conditions* if and only if it satisfies

$$\forall(p : p = p{\circ}\top\!\!\top : \top\!\!\top \subseteq p \Leftarrow R\backslash p \subseteq p) \quad . \tag{3.18}$$

□

Again we propose a definition in which the type difference between the variables is removed.

**Definition 3.19 (Relation Induction)** The relation $R$ is said to *admit relation induction* if and only if it satisfies

$$\forall(S ::\ \top\!\!\top \subseteq S \Leftarrow R\backslash S \subseteq S) \quad . \tag{3.20}$$

□

We prove the equivalence of all three definitions by "dualising" the proofs of theorems 3.7 and 3.8, i.e. we use the fact that Galois connected functions, in this case composition and factor, have what can be called "related" algebraic properties. The proofs may look more complicated than those of section 3.1 but they are not really.

**Theorem 3.21** For all relations $R$, that $R$ admits induction on monotypes equivales that $R$ admits induction on conditions.

**Proof** The proof is obtained by dualising the proof of theorem 3.7. With dummies $p$ and $A$ ranging over left conditions and monotypes, respectively, we have:

$$
\begin{array}{ll}
 & \forall(p :: \ \top\!\!\top \subseteq p \ \Leftarrow\ R\backslash p \subseteq p) \\
\equiv & \qquad \{ \ \text{range translation: } A \mapsto A\circ\top\!\!\top \text{ is a bijection} \ \} \\
 & \forall(A :: \ \top\!\!\top \subseteq A\circ\top\!\!\top \ \Leftarrow\ R\backslash(A\circ\top\!\!\top) \subseteq A\circ\top\!\!\top) \\
\equiv & \qquad \{ \ \text{condition-monotype isomorphism: (2.27) and} \\
 & \qquad\quad R\backslash(A\circ\top\!\!\top) = R\mathbin{\backslash\!\!\!\cdot}A\circ\top\!\!\top\text{, see below} \ \} \\
 & \forall(A :: \ I \subseteq A \ \Leftarrow\ R\mathbin{\backslash\!\!\!\cdot}A \subseteq A) \quad .
\end{array}
$$

That $R\backslash(A\circ\top\!\!\top) = R\mathbin{\backslash\!\!\!\cdot}A\circ\top\!\!\top$ is very easy to show from the defining Galois connections.

$$
\begin{array}{ll}
 & X \subseteq R\mathbin{\backslash\!\!\!\cdot}A\circ\top\!\!\top \\
\equiv & \qquad \{ \ \text{shunting: (2.20)} \ \} \\
 & X_{<} \subseteq R\mathbin{\backslash\!\!\!\cdot}A \\
\equiv & \qquad \{ \ \text{shunting: (2.37)} \ \} \\
 & (R\circ X_{<})_{<} \subseteq A \\
\equiv & \qquad \{ \ \text{domains: (2.31)} \ \} \\
 & (R\circ X)_{<} \subseteq A \\
\equiv & \qquad \{ \ \text{shunting: (2.20)} \ \} \\
 & R\circ X \subseteq A\circ\top\!\!\top \\
\equiv & \qquad \{ \ \text{shunting: (2.6)} \ \} \\
 & X \subseteq R\backslash(A\circ\top\!\!\top)
\end{array}
$$

□

**Theorem 3.22** For all relations $R$, that $R$ admits induction on conditions equivales that $R$ admits induction on relations.

**Proof** It is obvious that $R$ admits induction on conditions whenever it admits induction on relations (since every condition is also a relation). To prove the opposite implication assume that $R$ admits induction on conditions. Then, for all $S$ we have:

$$
\begin{array}{ll}
 & \top\!\!\top \subseteq S \\
\Leftarrow & \qquad \{ \ \top\!\!\top \subseteq \top\!\!\top\circ\top\!\!\top\text{, factor shunting} \ \} \\
 & \top\!\!\top \subseteq S/\top\!\!\top
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \{\quad S/\top\!\!\top = S/\top\!\!\top \circ \top\!\!\top \quad \text{, assumption: } R \text{ admits induction} \\
& \quad\ \text{on left conditions: (3.18)} \quad \} \\
& R\backslash(S/\top\!\!\top) \subseteq S/\top\!\!\top \\
\equiv & \{\quad \text{associativity of } \backslash \text{ and } / \text{ (2.9)} \quad \} \\
& (R\backslash S)/\top\!\!\top \subseteq S/\top\!\!\top \\
\Leftarrow & \{\quad \text{monotonicity of } /\top\!\!\top \quad \} \\
& R\backslash S \subseteq S \quad .
\end{array}
$$

□

Now the equivalence of the three formalisations of admitting induction has been established. Therefore, we will now speak only of "admitting induction" rather than "admitting monotype induction" or "admitting relation induction". And we can give yet another formulation, one in terms of fixed points. By an argument similar to the one preceeding definition 3.9 it follows that the fact that $R$ admits induction on monotypes can also be expressed as in the next definition.

**Definition 3.23 (Admitting induction)** Relation $R$ admits induction is equivalent to: $\mu(R\searrow) = I \quad .$
□

Just as was the case for well-foundedness, the definition in terms of fixed points immediately suggest a lemma. Because the monotype factor is anti-monotone in its left argument and by the monotonicity properties of the least fixed point operator we have that any relation which is contained in a relation that admits induction also has the property that it admits induction.

**Lemma 3.24** If relation $R$ admits induction and if $S \subseteq R$ then relation $S$ admits induction.
□

One might wonder why we break the symmetry between this section and section 3.1 and choose to express the definition of admitting induction on monotypes as a fixed point property. After all we can equally well say that admitting induction on relations is equivalent to $\mu(R\backslash) = \top\!\!\top$ . Both $\mu(R\backslash)$, which is a (left) condition, and $\mu(R\searrow)$ represent the same set. This set has an interesting property: any relation $R$ can be turned into a relation that admits induction by restricting the right domain to elements from the set represented by the monotype $\mu(R\searrow)$. And this is the reason why the monotype representation is preferred over the condition representation: as was argued in chapter 2, it is more convenient for our purposes to restrict the domain of a relation by composing it with a monotype than by intersecting it by a condition. That relations that admit induction can be constructed by restricting their right domain is formulated and proved in the following.

**Theorem 3.25** For any relation $R$ the relation $R \circ \mu(R\searrow)$ admits induction.

**Proof** We use 3.23 as definition of admitting induction: with $A$ denoting $\mu((R \circ \mu(R\searrow))\searrow)$ we must show $I = A$, or equivalently (because $A$ is a monotype) $I \subseteq A$. The proof has two phases. In the first phase we reduce the goal to the formally weaker $\mu(R\searrow) \subseteq A$.

$$
\begin{array}{ll}
 & I \subseteq A \\
\equiv & \quad \{ \text{ definition of } A,\ \mu F = F.\mu F \ \} \\
 & I \subseteq (R \circ \mu(R\searrow))\searrow A \\
\equiv & \quad \{ \text{ monotype factors } \} \\
 & (R \circ \mu(R\searrow))_< \subseteq A \\
\equiv & \quad \{ \ \mu(R\searrow) \text{ is a monotype } \} \\
 & (R \circ \mu(R\searrow) \circ \mu(R\searrow))_< \subseteq A \\
\equiv & \quad \{ \text{ monotype factors; definition of } A;\ \mu F = F.\mu F \ \} \\
 & \mu(R\searrow) \subseteq A
\end{array}
$$

Now, in the second phase, we have the opportunity to apply Knaster-Tarski.

$$
\begin{array}{ll}
 & \mu(R\searrow) \subseteq A \\
\Leftarrow & \quad \{ \text{ Knaster-Tarski } \} \\
 & R\searrow A \subseteq A \\
\equiv & \quad \{ \text{ definition of } A;\ \mu F = F.\mu F \ \} \\
 & R\searrow A \subseteq (R \circ \mu(R\searrow))\searrow A \\
\Leftarrow & \quad \{ \text{ monotype factor is anti-monotone in left argument } \} \\
 & R \circ \mu(R\searrow) \subseteq R \\
\equiv & \quad \{ \ \mu(R\searrow) \subseteq I;\ \text{monotonicity of composition } \} \\
 & \text{true}
\end{array}
$$

□

## 3.4 "Admits induction" and the transitive closure

The theorem comparable to theorem 3.11 is the following:

**Theorem 3.26** For all relations $R$, that $R$ admits induction equivales that $R^+$ admits induction.

**Proof** Again one implication is a straightforward consequence of lemma 3.24: since $R \subseteq R^+$ it follows that if $R^+$ admits induction then $R$ admits induction.

It remains to prove the converse. As for well-foundedness, we prove the more general statement that if $\mathsf{R}$ admits induction and $\mathsf{T}$ is related to $\mathsf{R}$ by the property

$$\mathsf{T}\circ\mathsf{T} \subseteq \mathsf{T}\circ\mathsf{R} \tag{3.27}$$

then $\mathsf{T}$ admits induction. The required result follows because $\mathsf{R}^+$ is indeed so related to $\mathsf{R}$. Assume (3.20). Assume $\mathsf{S}$ is an arbitrary relation. Then we aim to prove

$$\top\!\!\top \subseteq \mathsf{S} \Leftarrow \mathsf{T}\backslash\mathsf{S} \subseteq \mathsf{S}$$

for $\mathsf{T}$ satisfying (3.27). We have:

$$
\begin{array}{ll}
 & \top\!\!\top \subseteq \mathsf{S} \\
\Leftarrow & \qquad \{ \ \text{assumption: } \mathsf{T}\backslash\mathsf{S} \subseteq \mathsf{S}\ \text{, transitivity} \ \} \\
 & \top\!\!\top \subseteq \mathsf{T}\backslash\mathsf{S} \\
\Leftarrow & \qquad \{ \ (3.20),\ \mathsf{S} := \mathsf{T}\backslash\mathsf{S} \ \} \\
 & \mathsf{R}\backslash(\mathsf{T}\backslash\mathsf{S}) \subseteq \mathsf{T}\backslash\mathsf{S} \\
\Leftarrow & \qquad \{ \ \mathsf{T}\backslash\mathsf{S} \subseteq \mathsf{S}\ \text{, monotonicity and transitivity} \ \} \\
 & \mathsf{R}\backslash(\mathsf{T}\backslash\mathsf{S}) \subseteq \mathsf{T}\backslash(\mathsf{T}\backslash\mathsf{S}) \\
\equiv & \qquad \{ \ \text{pseudo-associativity of } \backslash : (2.8) \ \} \\
 & (\mathsf{T}\circ\mathsf{R})\backslash\mathsf{S} \subseteq (\mathsf{T}\circ\mathsf{T})\backslash\mathsf{S} \\
\Leftarrow & \qquad \{ \ \text{anti-monotonicity of } \backslash\mathsf{S} \ \} \\
 & \mathsf{T}\circ\mathsf{R} \supseteq \mathsf{T}\circ\mathsf{T} \quad .
\end{array}
$$

□

This concludes this section. We have established the equivalence of

- $\mathsf{R}$ admits monotype induction
- $\mathsf{R}$ admits condition induction
- $\mathsf{R}$ admits relation induction
- $\mu(\mathsf{R}\backslash\_) = \mathrm{I}$
- $\mathsf{R}^+$ admits induction.

## 3.5 Admits-induction implies well-founded

Now that we have seen several equivalent definitions of well-founded and admits-induction it is time to explore their relationship. The following lemma is the key insight.

**Lemma 3.28** $\nu(\circ R) \circ \mu(R\backslash) = \perp\!\!\perp$ .

**Proof** The form of the theorem suggests that we try to apply $\mu$-fusion. Of course then we have to find a suitable function $f$ such that $\mu f$ equals $\perp\!\!\perp$. Because it is formally weaker (but equivalent) we prove an inclusion, rather than equality. Furthermore we let $X$ denote $\nu(\circ R)$.

$$
\begin{array}{ll}
 & X \circ \mu(R\backslash) \subseteq \perp\!\!\perp \\
\Leftarrow & \quad \{ \ \mu\text{-fusion (2.4), for any } A. \text{ assume: } \mu f = \perp\!\!\perp \ \} \\
 & X \circ R\backslash A \subseteq f.(X \circ A) \\
\equiv & \quad \{ \ \text{by assumption } X \text{ is a fixed point of } \circ R \ \} \\
 & X \circ R \circ R\backslash A \subseteq f.(X \circ A) \\
\Leftarrow & \quad \{ \ \text{monotype factors (2.43), monotonicity and transitivity} \ \} \\
 & X \circ A \circ R \subseteq f.(X \circ A)
\end{array}
$$

So we can take the function $\circ R$ for $f$ provided that $\mu f = \perp\!\!\perp$. But this is true since $\perp\!\!\perp$ is the zero of composition. This completes the proof.
□

**Theorem 3.29** If $R$ admits induction then $R$ is well-founded.

**Proof** If $R$ admits induction then, by definition, $\mu(R\backslash) = I$ . So, by lemma 3.28, $\nu(\circ R) \circ I = \perp\!\!\perp$ . By definition 3.9 we have thus established that $R$ is well-founded.
□

One might suppose that "dualising" the above leads to a proof that well-foundedness implies admits-induction. Unfortunately this is not the case: complementation is needed to do that. We shall only sketch the proof since the equivalence between well-foundedness and admitting induction in a complemented relation algebra is well-known. To prove the theorem it suffices to know that in a complemented relation algebra $R\backslash A$ is the same as $\sim((\sim A \circ R)>)$ (where $\sim A$ is the complement relative to I, i.e. $\neg A \cap I$ ). This fact can then be used to show that $\sim((\nu(\circ R))>)$ is a fixed point of the function $(A \mapsto R\backslash A)$. Therefore, if $R$ admits induction $(\nu(\circ R))>$ equals I, hence $\top\!\!\top \circ \nu(\circ R) = \top\!\!\top$. To finish the proof one shows that $\nu(\circ R)$ is a right condition.

It is thus indeed the case that when we have complementation the sets represented by the condition $\nu(\circ R)$ and the monotype $\mu(R\backslash)$ are each other's complements. This also enables one to interpret $\mu(R\backslash)$. Because the condition $\nu(\circ R)$ is the set of all points from which an infinite $R$-chain starts, monotype $\mu(R\backslash)$ can be interpreted as the set of all points such that all chains starting in such a point are finite. We work in a complemented relation algebra and therefore we will from now on (in informal arguments only) use well-founded and admitting-induction interchangeably. It should be noted, however, that none of the results in this thesis depend on the equality of well-founded and admitting-induction.

## 3.6 Bound functions

As we will see in chapter 5 one of the ingredients of the correctness proof of repetitions is to show the well-foundedness of the body of the repetition. In practice this is often done by a technique that uses a bound function. This technique boils down to proving that there exists a function $f$ into a domain ordered by a well-founded relation $R$. The extra requirements are that $x\langle S\rangle y$ implies $f.x\langle R\rangle f.y$, or formulated in a point-free way: $S \subseteq f\cup \circ R \circ f$ . Here $S$ is to be interpreted as the input-output relation of the body (recall that input is from the right). That this is indeed enough to guarantee that relation $S$ is well-founded is because of the following theorem.

**Theorem 3.30 (Bound functions)** If $R$ is a relation and $f$ a functional relation such that $R \circ f<$ is well-founded, then relation $f\cup \circ R \circ f$ is well-founded as well.

**Proof** By definition 3.9 we have to show that the greatest fixed point of the function $X \mapsto X \circ f\cup \circ R \circ f$ equals $\perp\!\!\perp$. We do this by showing that any fixed point does so. Let relation $X$ satisfy $X = X \circ f\cup \circ R \circ f$. We first observe that such an $X$ has the property $X \circ f> = X$ because of $f \circ f> = f$. Thus,

$$\begin{array}{ll}
 & \text{true} \\
\equiv & \quad \{ \text{ assumption } \} \\
 & X \subseteq X \circ f\cup \circ R \circ f \\
\equiv & \quad \{ \text{ above observation and domains } \} \\
 & X \circ f> \subseteq X \circ f\cup \circ R \circ f< \circ f \\
\equiv & \quad \{ \text{ f is functional: definition 2.32 } \} \\
 & X \circ f\cup \subseteq X \circ f\cup \circ R \circ f< \\
\Rightarrow & \quad \{ \ R \circ f< \text{ is well-founded } \} \\
 & X \circ f\cup \subseteq \perp\!\!\perp \\
\equiv & \quad \{ \text{ f is functional: definition 2.32 } \} \\
 & X \circ f> \subseteq \perp\!\!\perp \circ f \\
\equiv & \quad \{ \text{ above observation; } \perp\!\!\perp \text{ is zero of composition } \} \\
 & X \subseteq \perp\!\!\perp
\end{array}$$

□

## 3.7 The lexical coupling

A standard way of constructing well-founded relations is the lexical coupling of two such relations. This lexical coupling of relations $\mathsf{R}$ and $\mathsf{S}$ is a relation on pairs, $\mathsf{X}$ say, with the property that either the first component is "decreased" by $\mathsf{R}$, or the second component is "decreased" by $\mathsf{S}$ while the first component remains unchanged:

$$(x,y)\langle \mathsf{X}\rangle(z,t) \quad \equiv \quad x\langle \mathsf{R}\rangle z \;\vee\; (x = z \wedge y\langle \mathsf{S}\rangle t) \tag{3.31}$$

In a point-free expression this becomes:

$$\mathsf{X} \;=\; \mathsf{R}\times\top\!\!\top \,\cup\, \mathrm{I}\times\mathsf{S} \tag{3.32}$$

In the special case that $\mathsf{R}$ and $\mathsf{S}$ are equal ordering relations, relation $\mathsf{X}$ is the lexicographic order on pairs of elements.

We now construct a proof of the statement that the lexical coupling of two well-founded relations is also well-founded. It follows from theorem 3.30 of the previous section that $\mathsf{R}\times\top\!\!\top$ is well-founded whenever $\mathsf{R}$ is: observe that $\ll\cup\circ\mathsf{R}\circ\ll$ equals $\mathsf{R}\times\top\!\!\top$, where $\ll$ is the projection function on the left coordinate. "Operationally", both relate the first components of their input and output by $\mathsf{R}$, while the only thing we know about the second components is that they are related by the total relation.

Similarly, $\top\!\!\top\times\mathsf{S}$ is well-founded whenever $\mathsf{S}$ is, and therefore also $\mathrm{I}\times\mathsf{S}$ because it is contained in $\top\!\!\top\times\mathsf{S}$. It now remains to show that the union of the two well-founded relations $\mathsf{R}\times\top\!\!\top$ and $\mathrm{I}\times\mathsf{S}$ is well-founded. It is certainly not generally the case that the union of two well-founded relations is well-founded. However, we have the following lemma.

**Lemma 3.33** If $\mathsf{R}$ and $\mathsf{S}$ are well-founded relations such that $\mathsf{R}\circ\mathsf{S}\subseteq\mathsf{R}$ then the relation $\mathsf{R}\cup\mathsf{S}$ is well-founded
□

We do not give a proof of this lemma here because in chapter 4 a much more general result will be derived. That the lexical coupling of $\mathsf{R}$ and $\mathsf{S}$ is well-founded now follows from lemma 3.33 and the next calculation.

$$\begin{array}{ll}
 & \mathsf{R}\times\top\!\!\top \circ \mathrm{I}\times\mathsf{S} \\
= & \qquad \{ \;\; \text{composition distributes over } \times \;\; \} \\
 & (\mathsf{R}\circ\mathrm{I})\times(\top\!\!\top\circ\mathsf{S}) \\
\subseteq & \qquad \{ \;\; \top\!\!\top\circ\mathsf{S}\subseteq\top\!\!\top;\ \times \text{ is monotone} \;\; \} \\
 & \mathsf{R}\times\top\!\!\top
\end{array}$$

So we have proved the lemma:

**Lemma 3.34** The lexical coupling $\mathsf{R}\times\top\!\!\top \cup \mathrm{I}\times\mathsf{S}$ of two relations $\mathsf{R}$ and $\mathsf{S}$ is well-founded if both $\mathsf{R}$ and $\mathsf{S}$ are well-founded.
□

It is also possible to show the converse: if the lexical coupling of two relations is well-founded the component relations are well-founded as well. First observe that if relation $\mathsf{R}{\times}\top\!\top \cup \mathrm{I}{\times}\mathsf{S}$ is well-founded, this is also true for both the smaller relations $\mathsf{R}{\times}\mathrm{I}$ and $\mathrm{I}{\times}\mathsf{S}$. So, we are done if we can show that well-foundedness of $\mathsf{R}$ follows from well-foundedness of $\mathsf{R}{\times}\mathrm{I}$. (By symmetry it then follows that $\mathsf{S}$ is well-founded.) By definition 3.5 we have to show $\mathsf{X}\subseteq\mathsf{X}{\circ}\mathsf{R} \Rightarrow \mathsf{X}\subseteq \perp\!\!\perp$ and we may use the assumption that $\mathsf{R}{\times}\mathrm{I}$ is well-founded.

$$
\begin{array}{ll}
 & \mathsf{X}\subseteq\mathsf{X}{\circ}\mathsf{R} \\
\Rightarrow & \qquad \{\ \ \times \text{ is a relator, hence it is monotone } \} \\
 & \mathsf{X}{\times}\mathrm{I}\subseteq(\mathsf{X}{\circ}\mathsf{R}){\times}\mathrm{I} \\
\equiv & \qquad \{\ \ \times \text{ is a relator, hence it distributes over composition } \} \\
 & \mathsf{X}{\times}\mathrm{I}\subseteq\mathsf{X}{\times}\mathrm{I}\circ\mathsf{R}{\times}\mathrm{I} \\
\Rightarrow & \qquad \{\ \ \text{by assumption } \mathsf{R}{\times}\mathrm{I} \text{ is well-founded definition 3.5 } \} \\
 & \mathsf{X}{\times}\mathrm{I}\subseteq\perp\!\!\perp
\end{array}
$$

From the interpretation, it is clear that $\mathsf{X}{\times}\mathrm{I}\subseteq\perp\!\!\perp$ implies that $\mathsf{X}\subseteq\perp\!\!\perp$, but it can also be proved calculationally as follows.

$$
\begin{array}{ll}
 & \mathsf{X}{\times}\mathrm{I}\subseteq\perp\!\!\perp \\
\Rightarrow & \qquad \{\ \ \text{monotonicity of composition } \} \\
 & \ll\circ\mathsf{X}{\times}\mathrm{I}\subseteq\ll\circ\perp\!\!\perp \\
\equiv & \qquad \{\ \ \text{property of product: } \ll\circ\mathsf{R}{\times}\mathsf{S} = \mathsf{R}\circ\ll\circ\,\mathrm{I}\times\mathsf{S}{>}; \\
 & \qquad\quad \perp\!\!\perp \text{ is zero of composition; } \mathrm{I}{>} = \mathrm{I} \ \} \\
 & \mathsf{X}{\circ}\ll\subseteq\perp\!\!\perp \\
\equiv & \qquad \{\ \ \text{domains: } \mathsf{R}{=}\perp\!\!\perp \equiv \mathsf{R}{<}{=}\perp\!\!\perp \ \} \\
 & (\mathsf{X}{\circ}\ll){<}\subseteq\perp\!\!\perp \\
\equiv & \qquad \{\ \ \text{domains: (2.31) } \} \\
 & (\mathsf{X}\circ\ll{<}){<}\subseteq\perp\!\!\perp \\
\equiv & \qquad \{\ \ \text{projections are surjective: } \ll{<} = \mathrm{I} \ \} \\
 & \mathsf{X}{<}\subseteq\perp\!\!\perp \\
\equiv & \qquad \{\ \ \text{domains: } \mathsf{R}{=}\perp\!\!\perp \equiv \mathsf{R}{<}{=}\perp\!\!\perp \ \} \\
 & \mathsf{X}\subseteq\perp\!\!\perp
\end{array}
$$

Summarising we have proved the following result, with which we conclude this chapter.

**Theorem 3.35 (Lexical coupling)** The lexical coupling $\mathsf{R}{\times}\top\!\top \cup \mathrm{I}{\times}\mathsf{S}$ is well-founded if and only if both $\mathsf{R}$ and $\mathsf{S}$ are well-founded.
□

# Chapter 4

# Example Reductivity Arguments

In the previous chapter we studied the notions "well-founded" and "admits induction". As we remarked there one of the goals was to start the development of a *calculational* theory that should be useful in the construction of calculational proofs. To see whether this theory can indeed be used we proceed in the present chapter to tackle the construction of a number of such proofs. Some of them use well-foundedness arguments, i.e. they use that a relation is well-founded; others depend on inductive arguments: they use that a relation admits induction. For reasons that will become clear in chapter 6 we use the term reductivity as an alternative for either well-foundedness or "admitting induction". Therefore, the proofs in this chapter can be called reductivity arguments.

Because the main purpose is to test the theory developed thus far, we decided that the theorems should be not too trivial. For instance, we give a proof of Newman's lemma [74], a lemma that is much exploited in the theory of term rewriting systems and is regarded as difficult to prove (as evidenced by the fact that it has been used to demonstrate the power of theorem proving systems [32]). We hope to demonstrate that the proof of the lemma becomes almost straightforward by reducing it to purely syntactic considerations.

Although the starting point of the proofs is in all cases a known theorem, it turned out that most proofs we constructed were in fact proofs of more general theorems. Sometimes, as was the case for the proof of Newman's lemma, the generalisation was only a minor one. However in another example we obtained a result that was substantially stronger than the theorem we tried to prove. This theorem of Geser [45] that served as the starting point gives a sufficient condition for the union of two well-founded relations to be well-founded. After the theorem was proved it turned out that we had a proof of a theorem which generalises both this result of Geser and a result of Bachmair and Dershowitz [3]. The generalisation was a by-product of the construction of the proof: beforehand, we were not even aware of the validity of the result of Bachmair and Dershowitz.

The proofs of this chapter are carried out in great detail. The reason is, again, that we want to test our theory. Therefore we do not want to shorten the calculations by relying on interpretation too much: we want to do real calculations where the steps we take are those that are prescribed by our theory. Those steps tend to be rather small and as a result the calculations tend to be rather lengthy, especially those in the later sections of this chapter. Because the steps are small and many, a disadvantage of our proofs might be that it is hard to find the most important steps or "milestones". To overcome this some of the longer calculations are split into smaller ones at the places where such a milestone occurs.

Some final remarks before we begin. The calculational theory of well-foundedness is still new and we do not have too much experience with it yet. Hence, when we use this theory to prove a theorem it can not be expected that the first proof that is found is the shortest or the best. However, we decided to present the proofs more or less as they were constructed, so they are not polished afterwards to make them shorter and more beautiful. We did not do the polishing because we want to show as much of the process of proof construction as possible. After all, this is what the theory is meant for: to be able to *construct* proofs, preferably in a beautiful way. It is not primarily intended to *present* results in a beautiful way, although we hope that it is evident that this is possible too.

We try to include some of the heuristics that lead to the discovery of the proofs. And, of course, during the construction of difficult proofs one occasionally encounters a dead end. Those are sometimes also indicated.

## 4.1 The uep of regular algebra

Definition 3.5 was introduced in [10] where it was called "definiteness". Reference [10] was about applying regular algebra to path-finding problems, and a fundamental fact exploited in that paper was that the property of being a regular algebra is preserved by matrix formation. Regular algebra is basically relation algebra without the converse operator and with some additional axioms. For instance, Salomaa's axiomatisation [79] of regular algebra involved the use of the so-called "empty word property". However, the formulation of this rule does not extend to matrices. As a replacement for Salomaa's rule the following rule was postulated in [10] as an axiom of regular algebra:

$$\mathsf{R} \text{ satisfies } (3.9) \equiv \forall(\mathsf{S},\mathsf{T} :: \mathsf{T} = \mathsf{S} \cup \mathsf{T}{\circ}\mathsf{R} \;\equiv\; \mathsf{T} = \mathsf{S}{\circ}\mathsf{R}^*) \;\; .$$

We call this rule the *unique extension property (uep) of regular algebra*. (In fact only an implication was postulated in [10]. As we see below the follows-from is very straightforward; it is also of lesser importance.)

In [6] it was shown that the rule remains valid for matrices by induction on the size of the matrix. Here we shall show that the rule is valid for any algebra complying with the properties detailed in sections 2.3.1 and 2.3.2 (the lattice and monoid structures of a relation algebra), and therefore for any regular algebra.

To make the discussion more precise we introduce the following definition.

**Definition 4.1** Relation $\mathsf{R}$ is said to be uniquely extendable iff it satisfies

$$\forall(\mathsf{S},\mathsf{T} :: \mathsf{T} = \mathsf{S} \cup \mathsf{T}{\circ}\mathsf{R} \;\equiv\; \mathsf{T} = \mathsf{S}{\circ}\mathsf{R}^*) \;\; . \tag{4.2}$$

□

Our claim is that unique-extendability and well-foundedness are equivalent properties.

That unique-extendability implies well-foundedness is not difficult to see. Instantiating $S$ in (4.2) with $\perp\!\!\perp$ we obtain (recall $\perp\!\!\perp$ is the unit of union and the zero of composition)

$$\forall(T :: T = T \circ R \quad \equiv \quad T = \perp\!\!\perp) \tag{4.3}$$

Next instantiating $T$ with $\nu(\circ R)$ and observing that in this case $T = T \circ R$, $\nu(\circ R) = \perp\!\!\perp$: i.e., by definition 3.9, $R$ is well-founded. This is the elementary part of the proof. The harder part is the reverse implication.

In words, definition 4.1 states that the function $T \mapsto S \cup T \circ R$ has a unique fixed point: $S \circ R^*$ . Now we observe that $S \circ R^*$ is the least fixed point of this function. Next, we show in a more general context that definition 4.1 is implied by the fact that the greatest and least fixed points of the function $T \mapsto S \cup T \circ R$ are the same.

**Lemma 4.4** If $f$ is a monotone function on a complete lattice with the property $\mu f = \nu f$, the following holds:

$$\forall(x :: \; x = f.x \quad \equiv \quad x = \mu f)$$

**Proof**

$$\begin{array}{ll}
 & x = f.x \\
\Rightarrow & \qquad \{ \quad \text{fixed points} \quad \} \\
 & \mu f \sqsubseteq x \wedge x \sqsubseteq \nu f \\
\equiv & \qquad \{ \quad \text{assumption: } \mu f = \nu f \quad \} \\
 & \mu f \sqsubseteq x \wedge x \sqsubseteq \mu f \\
\Rightarrow & \qquad \{ \quad \text{antisymmetry} \quad \} \\
 & x = \mu f \\
\Rightarrow & \qquad \{ \quad \text{fixed points} \quad \} \\
 & x = f.x
\end{array}$$

□

So we indeed are done if we can prove $\nu(T \mapsto S \cup T \circ R) = \mu(T \mapsto S \cup T \circ R)$.

We want to relate this to $R$ being well-founded —i.e. the greatest fixed point of the function $T \mapsto T \circ R$ is $\perp\!\!\perp$. In general we have $\nu(T \mapsto S \cup T \circ R) \supseteq \nu(T \mapsto T \circ R)$ since $\nu$ is a monotone function. So, there exist a relation $X$ such that $\nu(T \mapsto S \cup T \circ R) = X \cup \nu(T \mapsto T \circ R)$. If we can now prove that we are allowed to substitute $S \circ R^*$ for $X$, it follows that the least and greatest fixed point of the function $T \mapsto S \cup T \circ R$ are the same. To show that the substitution is allowed we construct such an $X$. We have:

$$\begin{array}{ll}
 & X \cup \nu(T \mapsto T \circ R) = \nu(T \mapsto S \cup T \circ R) \\
\Leftarrow & \qquad \{ \quad \nu\text{-fusion (the dual of } \mu\text{-fusion (2.5)) with } f := (X \cup),
\end{array}$$

$$
\begin{array}{ll}
 & \qquad g := (T \mapsto T\circ R), \text{ and } h := (T \mapsto S\cup T\circ R)\ . \\
 & \qquad (X\cup) \text{ is by assumption universally} \\
 & \qquad \cap\text{-distributive and thus has a lower adjoint } \} \\
 & \forall (T :: X\cup T\circ R = S\cup (X\cup T)\circ R) \\
\equiv & \qquad \{ \text{ distributivity } \} \\
 & \forall (T :: X\cup T\circ R = S\cup X\circ R\cup T\circ R) \\
\equiv & \qquad \{ \Rightarrow\text{: instantiate } T := \perp\!\!\!\perp \\
 & \qquad \Leftarrow\text{: Leibniz } \} \\
 & X = S\cup X\circ R\ .
\end{array}
$$

The least relation $X$ satisfying $X = S\cup X\circ R$ is $S\circ R^*$ therefore we conclude:

$$S\circ R^* \cup \nu(T \mapsto T\circ R) = \nu(T \mapsto S\cup T\circ R)\ . \tag{4.5}$$

Now we summarise the above argument in the next lemma.

**Lemma 4.6** If $R$ is well-founded then $R$ is uniquely extendable.

**Proof**

$$
\begin{array}{ll}
 & R \text{ is well-founded} \\
\equiv & \qquad \{ \text{ definition 3.9 } \} \\
 & \nu(T \mapsto T\circ R) = \perp\!\!\!\perp \\
\Rightarrow & \qquad \{ \text{ (4.5) } \} \\
 & \nu(T \mapsto S\cup T\circ R) = S\circ R^* \\
\Rightarrow & \qquad \{ S\circ R^* = \mu(T \mapsto S\cup T\circ R) \text{ and lemma 4.4} \\
 & \qquad \text{with } f := (T \mapsto S\cup T\circ R) \ \} \\
 & \forall (T :: T = S\cup T\circ R \ \equiv\ T = S\circ R^*) \\
\equiv & \qquad \{ \text{ definition 4.1 } \} \\
 & R \text{ is uniquely extendable}\ .
\end{array}
$$

□

Summarising, we have proved:

**Theorem 4.7** That $R$ is well-founded equivales that $R$ is uniquely-extendable.
□

Unique extendability is an attractive way of expressing well-foundedness. It is formally stronger than definition (3.9) since that definition is obtained by instantiating $\mathsf{S}$ to $\perp\!\!\perp$. It also illustrates clearly and succinctly a reason why well-foundedness is a useful attribute of a relation: in comparison to the definition of $\mathsf{R}^*$ which gives one a mechanism for proving only an *inclusion* of $\mathsf{S}{\circ}\mathsf{R}^*$ in a relation $\mathsf{T}$, well-foundedness of $\mathsf{R}$ gives one a mechanism for proving *equality* of a relation $\mathsf{T}$ with $\mathsf{S}{\circ}\mathsf{R}^*$.

## 4.2 A fact from graph theory

In this section we develop a formal proof of the following fact from graph theory: in an acyclic graph the vertices reachable from set $\mathsf{A}$ coincide with the vertices reachable from the minimal elements of $\mathsf{A}$. To be able to give a calculational proof we translate this fact to point-free relation algebra. Observe that we can do that because a graph can be viewed as a relation, $\mathsf{R}$ say, over a finite universe (the vertices). That there exists an edge from vertex $x$ to vertex $y$ is then expressed as $x\langle\mathsf{R}\rangle y$.

Now we have to define the meaning of a graph (or a finite relation) being acyclic. Obviously, a cycle gives rise to an infinite path in the graph. But, conversely, an infinite path in a graph is a cycle (because the number of vertices is finite). Therefore, acyclicity is the same as the absence of infinite paths i.e. well-foundedness. Because well-foundedness is equivalent to "admitting induction", an alternative is to express the fact that a relation is acyclic as: the relation admits induction.

To define reachability we observe that vertex $x$ is reachable from a set of vertices $\mathsf{A}$ if there exists $y{\in}\mathsf{A}$ such that there is a path from $y$ to $x$. That there is a path from $y$ to $x$ can of course be expressed as $y\langle\mathsf{R}^*\rangle x$ , so reachability of $x$ from $\mathsf{A}$ becomes $\exists(y : y{\in}\mathsf{A} : y\langle\mathsf{R}^*\rangle x)$ or by definition of composition: $\exists(y :: y\langle\mathsf{A}{\circ}\mathsf{R}^*\rangle x)$. In the last expression we recognise the pointwise definition of the domain operator, the expression is equivalent to $x{\in}(\mathsf{A}{\circ}\mathsf{R}^*)_>$. The point-free definition of the set $r.\mathsf{A}$ of vertices reachable from a set of vertices $\mathsf{A}$ is therefore:

$$r.\mathsf{A} = (\mathsf{A}{\circ}\mathsf{R}^*)_> \tag{4.8}$$

That a vertex $x$ is a minimal element of a set of vertices $\mathsf{A}$ means that $x$ is an element of $\mathsf{A}$ and that, furthermore, there is no edge from a vertex in $\mathsf{A}$ to $x$. This is more formally expressed as $x{\in}\mathsf{A} \wedge \neg\exists(y : y{\in}\mathsf{A} : y\langle\mathsf{R}\rangle x)$. Alternatively, by again introducing the domain operator, as $x \in \mathsf{A}\cap{\sim}((\mathsf{A}{\circ}\mathsf{R})_>)$. As we did before, the complement in the lattice of monotypes is denoted by $\sim$: ${\sim}\mathsf{A} = \neg\mathsf{A}\cap \mathrm{I}$. Furthermore, because it looks nicer, the intersection is replaced by a composition (this is allowed because it is an intersection on monotypes). The set $m.\mathsf{A}$ of minimal elements of $\mathsf{A}$ is thus defined as:

$$m.\mathsf{A} = \mathsf{A}\circ{\sim}((\mathsf{A}{\circ}\mathsf{R})_>) \tag{4.9}$$

The formal statement of the fact that the vertices reachable from set $\mathsf{A}$ coincide with the vertices reachable from the minimal elements of $\mathsf{A}$ now becomes:

$$r.\mathsf{A} = r.(m.\mathsf{A}) \tag{4.10}$$

This we have to prove for all monotypes $A$. Of this equality one inclusion is easy. From the definition (4.8) it is clear that $r$ is a monotone function. Furthermore from (4.9) we see that $m.A$ is contained in $A$. Therefore $r.A \supseteq r.(m.A)$. It remains to prove the other inclusion. We can proceed by trying to exploit the properties of $r.A$, but this leads nowhere. Of course this is not so strange because we somewhere have to use the assumption of acyclicity, or well-foundedness. But which formulation of that concept is the best choice? What we have to prove is $r.A \subseteq r.(m.A)$, and the righthand side of that formula contains a reflexive transitive closure. This suggests that we use one of the results of the preceeding section. Furthermore, it turns out that the expression $m.A$ does not play a role. Therefore, we begin by deriving a condition implying $r.A \subseteq r.B$.

$$
\begin{array}{ll}
 & r.A \subseteq r.B \\
\equiv & \quad \{ \text{ definition reachables: (4.8) } \} \\
 & r.A \subseteq (B \circ R^*)> \\
\equiv & \quad \{ \text{ monotype-condition isomorphism } \} \\
 & \top\!\!\top \circ r.A \subseteq \top\!\!\top \circ (B \circ R^*)> \\
\equiv & \quad \{ \text{ domains: } \top\!\!\top \circ X = \top\!\!\top \circ X> \; \} \\
 & \top\!\!\top \circ r.A \subseteq \top\!\!\top \circ B \circ R^*
\end{array}
$$

Now we can use the acyclicity of $R$. From the discussion in the preceeding section on the uep of regular algebra it follows that, for well-founded $R$, relation $\top\!\!\top \circ B \circ R^*$ is the greatest fixed point of the function $X \mapsto (\top\!\!\top \circ B \cup X \circ R)$ (compare (4.5)). So we could try to continue the calculation by using Knaster-Tarski. However, this is a dead end and the problem is caused by the relation $R^*$ on the lefthand side. So we remove that one first.

$$
\begin{array}{ll}
 & \top\!\!\top \circ r.A \subseteq \top\!\!\top \circ B \circ R^* \\
\equiv & \quad \{ \text{ definition reachables: (4.8);} \\
 & \quad\;\; \text{domains: } \top\!\!\top \circ X = \top\!\!\top \circ X>; \\
 & \quad\;\; R^* = R^* \circ R^* \; \} \\
 & \top\!\!\top \circ A \circ R^* \subseteq \top\!\!\top \circ B \circ R^* \circ R^* \\
\Leftarrow & \quad \{ \text{ monotonicity } \} \\
 & \top\!\!\top \circ A \subseteq \top\!\!\top \circ B \circ R^* \\
\Leftarrow & \quad \{ \text{ } R \text{ is well-founded:} \\
 & \quad\;\; \top\!\!\top \circ B \circ R^* = \nu(X \mapsto (\top\!\!\top \circ B \cup X \circ R)); \\
 & \quad\;\; \text{Knaster-Tarski } \} \\
 & \top\!\!\top \circ A \subseteq \top\!\!\top \circ B \cup \top\!\!\top \circ A \circ R \\
\equiv & \quad \{ \text{ distribution of composition over union;}
\end{array}
$$

$$
\begin{array}{ll}
 & \qquad \text{domains: } \top\!\top\circ X = \top\!\top\circ X{>} \quad \} \\
 & \top\!\top\circ A \subseteq \top\!\top\circ(B\cup A\circ R){>} \\
\equiv & \qquad \{ \quad \text{monotype-condition isomorphism} \quad \} \\
 & A \subseteq (B\cup A\circ R){>} \\
\equiv & \qquad \{ \quad \text{domain operator is } \cup\text{-junctive} \quad \} \\
 & A \subseteq B\cup(A\circ R){>} \\
\equiv & \qquad \{ \quad \text{shunting in the monotype lattice} \quad \} \\
 & A\circ\sim((A\circ R){>}) \subseteq B \\
\equiv & \qquad \{ \quad \text{definition (4.9)} \quad \} \\
 & \mathfrak{m}.A \subseteq B
\end{array}
$$

With this calculation we have shown that if set $B$ contains the minimal elements of a set $A$ then the elements reachable from $A$ are reachable from $B$ too. More precisely:

$$\mathfrak{m}.A \subseteq B \quad\Rightarrow\quad \mathfrak{r}.A\subseteq\mathfrak{r}.B \quad . \tag{4.11}$$

Instantiating $B$ with $\mathfrak{m}.A$ in this formula gives then the desired result:

$$\mathfrak{r}.A\subseteq\mathfrak{r}.(\mathfrak{m}.A) \quad .$$

And this completes the proof of (4.10).

An interesting observation can be made if we take a closer look at the antecedent of formula (4.11). After instantiating $B$ to the empty relation and writing out the definition of $\mathfrak{m}$ it reads: $A\circ\sim((A\circ R){>}) \subseteq \perp\!\!\perp$ . Now we can apply shunting in the monotype lattice and we get $A \subseteq (A\circ R){>}$. This expression is the antecedent in definition 3.2. So, yet another formulation of a relation being well-founded is (for all monotypes $A$):

$$\mathfrak{m}.A \subseteq \perp\!\!\perp \quad\Rightarrow\quad A\subseteq\perp\!\!\perp \quad , \tag{4.12}$$

or the equivalent contrapositive (using that $\perp\!\!\perp$ is the bottom of the lattice):

$$A \neq \perp\!\!\perp \quad\Rightarrow\quad \mathfrak{m}.A \neq \perp\!\!\perp \quad . \tag{4.13}$$

This is the familiar characterisation “every non-empty set has a minimal element” of well-foundedness.

Now we can ask whether the converse of the fact that was proved in this section also holds. Is it true that a graph with property (4.10) does not contain a cycle? This question can be answered affirmatively and the proof is very simple. We show that a graph satisfying (4.10) also satisfies (4.12).

$$
\begin{array}{ll}
 & \mathfrak{m}.A = \perp\!\!\perp \\
\Rightarrow & \qquad \{ \quad \text{Leibniz} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
& r.(m.A) = r.\perp\!\!\!\perp \\
\equiv & \quad \{ \text{ assumption: } r.(m.A) = r.A; \\
& \quad\ \ \text{definition of } r, (4.8) \ \} \\
& r.A = (\perp\!\!\!\perp \circ R^*)> \\
\equiv & \quad \{ \text{ definition of } r, (4.8); \\
& \quad\ \ \perp\!\!\!\perp \text{ is zero of composition } \} \\
& (A \circ R^*)> = \perp\!\!\!\perp \\
\Rightarrow & \quad \{ \ I \subseteq R^* \ \} \\
& A \subseteq \perp\!\!\!\perp
\end{array}
$$

## 4.3 Newman's lemma

Newman's lemma [74] is a lemma from the study of term rewriting systems. These systems play an important role in, for instance, the implementation of functional programming languages. By definition, a term rewrite system is a set, together with a set of rewrite rules. A typical, and about theoldest, example is the $\lambda$-calculus: a set of terms, together with a set of rewrite rules such as $\beta$-reduction. The rewrite rules induce a relation on the terms of the system, so, reduced to its bare essentials, a term rewrite system is just a relation. Now an important property of the relation associated with a term rewrite system is that of confluence, another one is local confluence (also known as the Church-Rosser and weakly Church-Rosser properties respectively). For an account of term rewrite systems in general and these properties in particular see [58].

Newman's lemma states that any relation that admits induction and is locally confluent is also confluent. Because the point-wise formulations are, for the present discussion, not relevant we immediately give the point-free expressions. Relation $R$ is confluent is equivalent to

$$R^* \circ (R\cup)^* \subseteq (R\cup)^* \circ R^* \quad . \tag{4.14}$$

Relation $R$ is locally confluent can be formulated as:

$$R \circ R\cup \subseteq (R\cup)^* \circ R^* \quad . \tag{4.15}$$

See [15] for how one obtains these definitions from the usual formulations. Because it turns out that it is not essential for the proof of the lemma that the relations are each other's converse, we generalise (4.14) and (4.15) to

$$R^* \circ S^* \subseteq S^* \circ R^* \quad , \tag{4.16}$$

and

$$R \circ S \subseteq S^* \circ R^* \quad . \tag{4.17}$$

This generalisation of confluence is called commutation in the literature. Thus the first, (4.16) is a point free formulation of the fact that $R$ and $S$ commute, whereas the second (4.17) says that $R$ and $S$ commute locally. Using these terms we can now formulate the following generalisation of Newman's lemma.

**Lemma 4.18 (Newman)** If $R$ and $S$ are relations such that $R \cup S^{\cup}$ admits induction and $R$ and $S$ commute locally, then $R$ and $S$ commute.

**Proof** First we remark that we have, by the properties of monotype factors, specifically $(R \cup S)\searrow B = R\searrow B \circ S\searrow B$ and $R^{\cup}\searrow B = B\swarrow R$:

$$(R \cup S^{\cup})\searrow B = R\searrow B \circ B\swarrow S \quad . \tag{4.19}$$

Therefore the assumption that $R \cup S^{\cup}$ admits induction (on monotypes, definition 3.15) is equivalent to, for all monotypes $B$:

$$R\searrow B \circ B\swarrow S \subseteq B \Rightarrow I \subseteq B \quad . \tag{4.20}$$

We also have the assumption that $R$ and $S$ commute locally, i.e.

$$R \circ S \subseteq S^{*} \circ R^{*} \quad . \tag{4.21}$$

What we have to show is: $R^{*} \circ S^{*} \subseteq S^{*} \circ R^{*}$. We start our proof by a calculation which takes this expression into the form $I \subseteq B$, because this makes (4.20) applicable.

$$\begin{array}{ll}
 & R^{*} \circ S^{*} \subseteq S^{*} \circ R^{*} \\
\equiv & \{ \text{ I is identity of composition;} \\
 & \quad \text{we choose to do induction on the ``middle'' I to keep} \\
 & \quad \text{the symmetry } \} \\
 & R^{*} \circ I \circ S^{*} \subseteq S^{*} \circ R^{*} \\
\equiv & \{ \text{ (2.6) and (2.9) } \} \\
 & I \subseteq R^{*} \backslash (S^{*} \circ R^{*}) / S^{*} \\
\equiv & \{ \; I \subseteq I \; \} \\
 & I \subseteq R^{*} \backslash (S^{*} \circ R^{*}) / S^{*} \cap I \quad .
\end{array}$$

To reduce the length of the expressions we introduce the shorthand

$$A = R^{*} \backslash (S^{*} \circ R^{*}) / S^{*} \cap I \quad .$$

The previous calculation gives us also (replace in the lefthand sides of the last three lines $I$ by $B$) for an arbitrary monotype $B$,

$$B \subseteq A \quad \equiv \quad R^{*} \circ B \circ S^{*} \subseteq S^{*} \circ R^{*} \quad . \tag{4.22}$$

By instantiating $B$ in (4.22) to $A$ we obtain the cancellation property:

$$R^{*} \circ A \circ S^{*} \subseteq S^{*} \circ R^{*} \quad . \tag{4.23}$$

After this investigation of $A$, we continue the main calculation.

$$\mathrm{I} \subseteq \mathsf{R}^*\backslash(\mathsf{S}^*{\circ}\mathsf{R}^*)/\mathsf{S}^* \cap \mathrm{I}$$

$$\equiv \qquad \{ \quad \text{definition of } \mathsf{A} \quad \}$$

$$\mathrm{I} \subseteq \mathsf{A}$$

$$\Leftarrow \qquad \{ \quad (4.20) \quad \}$$

$$\mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \subseteq \mathsf{A}$$

$$\equiv \qquad \{ \quad (4.22) \quad \}$$

$$\mathsf{R}^* \circ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \circ \mathsf{S}^* \ \subseteq\ \mathsf{S}^*{\circ}\mathsf{R}^*$$

$$\equiv \qquad \{ \quad \mathsf{X}^*{\circ}\mathsf{Y}{\circ}\mathsf{Z}^* = \mathsf{X}^*{\circ}\mathsf{Y} \cup \mathsf{Y}{\circ}\mathsf{Z}^* \cup \mathsf{X}^*{\circ}\mathsf{X}{\circ}\mathsf{Y}{\circ}\mathsf{Z}{\circ}\mathsf{Z}^*; \ \text{see below.} \quad \}$$

$$\mathsf{R}^* \circ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \subseteq \mathsf{S}^*{\circ}\mathsf{R}^*$$

$$\wedge\ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \circ \mathsf{S}^* \ \subseteq\ \mathsf{S}^*{\circ}\mathsf{R}^*$$

$$\wedge\ \mathsf{R}^* \circ \mathsf{R} \circ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \circ \mathsf{S} \circ \mathsf{S}^* \ \subseteq\ \mathsf{S}^*{\circ}\mathsf{R}^* \quad .$$

The last step is motivated by the fact that it makes factor cancellation applicable. We have to admit that the rule, mentioned in the hint, was obtained by looking into the model. We also do not provide a proof of that rule: it is of the kind "unfold the definition, distribute and rearrange term, then fold the definition". Now the first two conjuncts of this last clause are true; we prove only the first one, the proof of the second proceeds similarly.

$$\mathsf{R}^* \circ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S}$$

$$\subseteq \qquad \{ \quad \text{I is unit of composition; } \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \subseteq \mathrm{I}{\circ}\mathrm{I} \subseteq \mathrm{I} \quad \}$$

$$\mathrm{I} \circ \mathsf{R}^* \circ \mathrm{I}$$

$$\subseteq \qquad \{ \quad \mathsf{S}^* \text{ is reflexive: } \mathrm{I} \subseteq \mathsf{S}^* \quad \}$$

$$\mathsf{S}^*{\circ}\mathsf{R}^* \quad .$$

So the remaining obligation is to prove the last conjunct.

$$\mathsf{R}^* \circ \mathsf{R} \circ \mathsf{R}\underline{\backslash}\mathsf{A} \circ \mathsf{A}\underline{/}\mathsf{S} \circ \mathsf{S} \circ \mathsf{S}^*$$

$$\subseteq \qquad \{ \quad \text{monotype factors: } \mathsf{R} \circ \mathsf{R}\underline{\backslash}\mathsf{A} \subseteq \mathsf{A} \circ \mathsf{R}, \text{ and the dual} \quad \}$$

$$\mathsf{R}^* \circ \mathsf{A} \circ \mathsf{R} \circ \mathsf{S} \circ \mathsf{A} \circ \mathsf{S}^*$$

$$\subseteq \qquad \{ \quad \text{assumption (4.21)} \quad \}$$

$$\mathsf{R}^* \circ \mathsf{A} \circ \mathsf{S}^* \circ \mathsf{R}^* \circ \mathsf{A} \circ \mathsf{S}^*$$

$$\subseteq \qquad \{ \quad (4.23) \quad \}$$

$$\mathsf{S}^* \circ \mathsf{R}^* \circ \mathsf{R}^* \circ \mathsf{A} \circ \mathsf{S}^*$$

$\subseteq \qquad \{ \ \mathsf{R}^* \text{ is transitive } \}$

$\mathsf{S}^* \circ \mathsf{R}^* \circ \mathsf{A} \circ \mathsf{S}^*$

$\subseteq \qquad \{ \ (4.23) \ \}$

$\mathsf{S}^* \circ \mathsf{S}^* \circ \mathsf{R}^*$

$\subseteq \qquad \{ \ \mathsf{S}^* \text{ is transitive } \}$

$\mathsf{S}^* \circ \mathsf{R}^*$ .

This completes the proof.
□

Compared to the original proof in [74] the proof given here is much simpler. But that might not be a fair comparison because Newman's proof is quite old. There are also more recent publications that present a proof of the lemma. Sometimes such a proof consists only of a picture and a few words such as in [14]. Here one could say that concision has been achieved at the expense of precision. Other proofs [70] depend on a rather complicated lemma on so-called "multiset extensions" of well-founded relations.

Our proof has the advantage that it depends on first principles only and is precise. A disadvantage is, maybe, that it is rather long and for some people not illuminating. Such people are referred to [55] for a proof that is comparable to the one given here. The proof there is not calculational but if its reasoning were brought into a calculational form it would look very much like our proof. However, it should be remarked that the generalisation — the replacement of $\mathsf{R}^\cup$ by an arbitrary relation— emerged quite naturally from the proof we constructed for Newman's lemma in its original form. This generalisation is in our opinion not so easy to see in a proof like the one in [55].

## 4.4 Modularity of well-foundedness

In general, it is not the case that the union of two well-founded relations is a well-founded relation itself. Or, to state this in the terminology of [70], well-foundedness is not a modular property of relations. The purpose of this section is to derive a condition on two well-founded relations such that their union is well-founded. Reference [45] gives transitivity of the union as such a condition, and for a long time we tried to find a calculational proof of this result. However, this turned out to be a very difficult task. When we finally managed to find a proof, it happened to be a proof of a much stronger result.

The reader is warned that the argument is very long and in a sense not very illuminating. Although we have tried to include some of the heuristics that lead to the proof we are aware of the fact that these are not totally convincing. But at the moment this is the best we can do.

The proof begins simply enough. We assume that relations $\mathsf{R}$ and $\mathsf{S}$ are well-founded. Our task is to derive additional conditions on $\mathsf{R}$ and $\mathsf{S}$ that guarantee that any relation $\mathsf{X}$

$$\mathsf{X} = \mathsf{X} \circ (\mathsf{R} \cup \mathsf{S}) \tag{4.24}$$

equals the empty relation. Because this worked well with most of the other proofs in this chapter, we exploit the fact that R and S are well-founded as soon as possible.

$$
\begin{array}{ll}
 & \mathsf{X} \subseteq \perp\!\!\perp \\
\Leftarrow & \quad \{ \ \mathsf{S} \text{ is well-founded } \} \\
 & \mathsf{X} \subseteq \mathsf{X} \circ \mathsf{S} \\
\equiv & \quad \{ \ \text{shunting } \} \\
 & \mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S}) \subseteq \perp\!\!\perp \\
\Leftarrow & \quad \{ \ \mathsf{R} \text{ is well-founded.} \\
 & \quad \quad \text{definition: } \mathsf{Y} = \mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S}) \ \} \\
 & \mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S}) \subseteq \mathsf{Y} \circ \mathsf{R}
\end{array}
$$

To make the calculations more compact we introduced the shorthand:

$$\mathsf{Y} = \mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S}) \tag{4.25}$$

This shorthand is not used in the lefthand side of the last clause because that expression can be rewritten a bit: by property (4.24) and distribution of composition over unions, $\mathsf{X}$ equals $\mathsf{X} \circ \mathsf{R} \cup \mathsf{X} \circ \mathsf{S}$. So, by absorption, $\mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S})$ equals $\mathsf{X} \circ \mathsf{R} \cap \neg(\mathsf{X} \circ \mathsf{S})$. The latter relation is formally smaller than the former and therefore this rewrite results in a (formally) weaker proof obligation.

$$
\begin{array}{ll}
 & \mathsf{X} \cap \neg(\mathsf{X} \circ \mathsf{S}) \subseteq \mathsf{Y} \circ \mathsf{R} \\
\equiv & \quad \{ \ \text{see above } \} \\
 & \mathsf{X} \circ \mathsf{R} \cap \neg(\mathsf{X} \circ \mathsf{S}) \subseteq \mathsf{Y} \circ \mathsf{R} \\
\equiv & \quad \{ \ \text{shunting } \} \\
 & \mathsf{X} \circ \mathsf{R} \subseteq \mathsf{Y} \circ \mathsf{R} \cup \mathsf{X} \circ \mathsf{S}
\end{array}
$$

Now we can use the fact that $\mathsf{Y}$ is contained in $\mathsf{X}$ to replace the occurrence of $\mathsf{X}$ in the righthand side by $\mathsf{Y}$. Then $\mathsf{Y}$ can be factored out, thereby creating the expression $\mathsf{R} \cup \mathsf{S}$. After that we might make the assumption $\mathsf{U} \subseteq \mathsf{R} \cup \mathsf{S}$, replace $\mathsf{R} \cup \mathsf{S}$ by $\mathsf{U}$ and see what conditions on $\mathsf{U}$ can be found such that the proof can be brought to a successful conclusion. But before we do all this it is good to see the next lemma.

**Lemma 4.26** For all relations $\mathsf{U}$ and $\mathsf{V}$, if $\mathsf{U} = \mathsf{U} \circ \mathsf{V}$ then $\mathsf{U} = \mathsf{U} \circ \mathsf{V}^*$

**Proof**

$$
\begin{array}{ll}
 & \mathsf{U} = \mathsf{U} \circ \mathsf{V}^* \\
\equiv & \quad \{ \ \mathrm{I} \subseteq \mathsf{V}^*, \text{ hence by monotonicity: } \mathsf{U} \subseteq \mathsf{U} \circ \mathsf{V}^* \ \} \\
 & \mathsf{U} \circ \mathsf{V}^* \subseteq \mathsf{U}
\end{array}
$$

$$\begin{array}{ll}
\Leftarrow & \{ \quad \mathsf{U}\circ\mathsf{V}^* \text{ is the least fixed point of the function} \\
& \quad \mathsf{Y} \mapsto \mathsf{U}\cup\mathsf{Y}\circ\mathsf{V}\text{; Knaster-Tarski} \quad \} \\
& \mathsf{U}\cup\mathsf{U}\circ\mathsf{V} \subseteq \mathsf{U} \\
\equiv & \{ \quad \text{assumption: } \mathsf{U} = \mathsf{U}\circ\mathsf{V} \quad \} \\
& \text{true}
\end{array}$$

□

So, we first replace $\mathsf{X}$ by $\mathsf{X}\circ(\mathsf{R}\cup\mathsf{S})^*$ because this again results in a formally weaker proof obligation.

$$\begin{array}{ll}
& \mathsf{X}\circ\mathsf{R} \subseteq \mathsf{Y}\circ\mathsf{R} \,\cup\, \mathsf{X}\circ\mathsf{S} \\
\equiv & \{ \quad \text{above lemma and assumption (4.24)} \quad \} \\
& \mathsf{X}\circ\mathsf{R} \subseteq \mathsf{Y}\circ\mathsf{R} \,\cup\, \mathsf{X}\circ(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S} \\
\Leftarrow & \{ \quad \mathsf{Y}\subseteq\mathsf{X} \quad \} \\
& \mathsf{X}\circ\mathsf{R} \subseteq \mathsf{Y}\circ\mathsf{R} \,\cup\, \mathsf{Y}\circ(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S} \\
\equiv & \{ \quad \text{distribution of composition over union} \quad \} \\
& \mathsf{X}\circ\mathsf{R} \subseteq \mathsf{Y}\circ(\mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S}) \\
\Leftarrow & \{ \quad \text{assumption: } \mathsf{U}\circ\mathsf{R} \subseteq \mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S} \quad \} \\
& \mathsf{X}\circ\mathsf{R} \subseteq \mathsf{Y}\circ\mathsf{U}\circ\mathsf{R} \\
\Leftarrow & \{ \quad \text{monotonicity of composition} \quad \} \\
& \mathsf{X} \subseteq \mathsf{Y}\circ\mathsf{U}
\end{array}$$

Here we made the assumption

$$\mathsf{U}\circ\mathsf{R} \subseteq \mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S} \tag{4.27}$$

to get rid of both the expression $\mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S}$ and the $\mathsf{R}$ on the lefthand side. This condition looks promising: if we replace $\mathsf{U}$ by $\mathsf{S}$ it generalises the condition $\mathsf{S}\circ\mathsf{R}\subseteq\mathsf{R}$, which is sufficient to guarantee well-foundedness of $\mathsf{R}\cup\mathsf{S}$. It also generalises the condition $\mathsf{S}\circ\mathsf{R}\subseteq(\mathsf{R}\cup\mathsf{S})^*\circ\mathsf{S}$, which is called “R and S quasi-commute” in the literature, and is sufficient to guarantee well-foundedness of the union (of course under the assumption that $\mathsf{R}$ and $\mathsf{S}$ are well-founded).

Recall that $\mathsf{Y}$ is defined as an intersection. To be able to do something with the expression $\mathsf{Y}\circ\mathsf{U}$ we therefore need a lemma relating composition and intersection. We have the rule $(\mathsf{R}\cap\mathsf{S})\circ\mathsf{T} \subseteq \mathsf{R}\circ\mathsf{T}\cap\mathsf{S}\circ\mathsf{T}$ but if we try to apply that one it turns out that the intersection “goes the wrong way around”. We need that by distributing the composition into the intersection the result becomes larger. But in this case we know something more about the intersection: it is the intersection of a relation and the complement of a relation. For this we have the following lemma.

**Lemma 4.28** For all relations $V$, $W$ and $Z$:

$$V \circ Z \cap \neg(W \circ Z) \subseteq (V \cap \neg W) \circ Z$$

**Proof**

$$
\begin{array}{ll}
 & V \circ Z \cap \neg(W \circ Z) \subseteq (V \cap \neg W) \circ Z \\
\equiv & \quad \{ \text{ shunting } \} \\
 & V \circ Z \subseteq (V \cap \neg W) \circ Z \cup W \circ Z \\
\equiv & \quad \{ \text{ distribution of composition over union } \} \\
 & V \circ Z \subseteq ((V \cap \neg W) \cup W) \circ Z \\
\equiv & \quad \{ \text{ absorption } \} \\
 & V \circ Z \subseteq (V \cup W) \circ Z \\
\equiv & \quad \{ \text{ monotonicity } \} \\
 & \text{true}
\end{array}
$$

□

So we can continue the main calculation and distribute the relation $U$ using this lemma. However, that does not help: it does not lead to a sensible condition on $U$. In other words we are stuck. It was difficult to find the way out. Surprisingly, it turned out to be necessary to use the well-foundedness of $S$ once more; and, in our opinion this is the biggest invention in the proof.

$$
\begin{array}{ll}
 & X \subseteq Y \circ U \\
\equiv & \quad \{ \text{ shunting } \} \\
 & X \cap \neg(Y \circ U) \subseteq \perp\!\!\perp \\
\Leftarrow & \quad \{ \text{ } S \text{ is well-founded } \} \\
 & X \cap \neg(Y \circ U) \subseteq (X \cap \neg(Y \circ U)) \circ S \\
\Leftarrow & \quad \{ \text{ above lemma } \} \\
 & X \cap \neg(Y \circ U) \subseteq X \circ S \cap \neg(Y \circ U \circ S) \\
\equiv & \quad \{ \text{ intersection } \} \\
 & X \cap \neg(Y \circ U) \subseteq X \circ S \\
 & \wedge \;\; X \cap \neg(Y \circ U) \subseteq \neg(Y \circ U \circ S)
\end{array}
$$

We now handle the last two conjuncts separately.

$$X \cap \neg(Y \circ U) \subseteq X \circ S$$

$$
\begin{array}{ll}
\equiv & \{\ \text{trading}\ \} \\
& X \cap \neg(X \circ S) \subseteq Y \circ U \\
\equiv & \{\ \text{definition of } Y\ \} \\
& Y \subseteq Y \circ U \\
\Leftarrow & \{\ \text{monotonicity of composition}\ \} \\
& I \subseteq U
\end{array}
$$

This calculation gives a first condition on $U$: it has to be reflexive. Next we take the other conjunct.

$$
\begin{array}{ll}
& X \cap \neg(Y \circ U) \subseteq \neg(Y \circ U \circ S) \\
\equiv & \{\ \text{trading}\ \} \\
& X \cap Y \circ U \circ S \subseteq Y \circ U \\
\Leftarrow & \{\ X \cap T \subseteq T\ \} \\
& Y \circ U \circ S \subseteq Y \circ U \\
\Leftarrow & \{\ \text{monotonicity of composition}\ \} \\
& U \circ S \subseteq U
\end{array}
$$

Combining the two conditions on $U$ we get: $I \cup U \circ S \subseteq U$. The least such $U$ is of course $S^*$ and substituting this for $U$ we obtain as a condition for well-foundedness of $R \cup S$:

$$S^* \circ R \subseteq R \cup (R \cup S)^* \circ S \tag{4.29}$$

We can do a bit better because it is possible to drop the star on the lefthand side as is shown in the final calculation.

$$
\begin{array}{ll}
& S^* \circ R \subseteq R \cup (R \cup S)^* \circ S \\
\Leftarrow & \{\ S^* \circ R \text{ is the least fixed point of the function} \\
& \quad Y \mapsto R \cup S \circ Y\text{; Knaster-Tarski}\ \} \\
& R \cup S \circ (R \cup (R \cup S)^* \circ S) \subseteq R \cup (R \cup S)^* \circ S \\
\Leftarrow & \{\ R \subseteq R \cup T\text{; distribution of composition over union}\ \} \\
& S \circ R \cup S \circ (R \cup S)^* \circ S \subseteq R \cup (R \cup S)^* \circ S \\
\equiv & \{\ \text{union}\ \} \\
& S \circ R \subseteq R \cup (R \cup S)^* \circ S \\
& \wedge\ S \circ (R \cup S)^* \cdot S \subseteq R \cup (R \cup S)^* \circ S \\
\Leftarrow & \{\ \text{monotonicity}\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{S}\circ\mathsf{R} \;\subseteq\; \mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^{*}\circ\mathsf{S} \\
 & \wedge\;\; \mathsf{S}\circ(\mathsf{R}\cup\mathsf{S})^{*}\cdot\mathsf{S} \;\subseteq\; (\mathsf{R}\cup\mathsf{S})^{*}\circ\mathsf{S} \\
\equiv & \qquad \{\;\; \mathsf{S}\subseteq\mathsf{R}\cup\mathsf{S};\ \mathsf{T}\circ\mathsf{T}^{*}\subseteq\mathsf{T}^{*} \;\;\} \\
 & \mathsf{S}\circ\mathsf{R} \;\subseteq\; \mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^{*}\circ\mathsf{S}
\end{array}
$$

Summarising, we have proved the following theorem.

**Theorem 4.30** Suppose that $\mathsf{R}$ and $\mathsf{S}$ are well-founded relations. That the union $\mathsf{R}\cup\mathsf{S}$ is well-founded follows from:

$$\mathsf{S}\circ\mathsf{R} \;\subseteq\; \mathsf{R}\cup(\mathsf{R}\cup\mathsf{S})^{*}\circ\mathsf{S}$$

□

This theorem has many corollaries. For instance we can use reflexivity of $(\mathsf{R}\cup\mathsf{S})^{*}$, i.e. replace this expression with I. Then we get the condition $\mathsf{S}\circ\mathsf{R}\subseteq\mathsf{R}\cup\mathsf{S}$, which is clearly implied by transitivity of $\mathsf{R}\cup\mathsf{S}$. Also the $\mathsf{R}\cup$-part can be dropped, giving $\mathsf{S}\circ\mathsf{R}\subseteq(\mathsf{R}\cup\mathsf{S})^{*}\circ\mathsf{S}$, which is called "$\mathsf{R}$ and $\mathsf{S}$ quasi commute". Two other obvious conditions are $\mathsf{S}\circ\mathsf{R}\subseteq\mathsf{R}$ and $\mathsf{S}\circ\mathsf{R}\subseteq\mathsf{S}$ the latter already having been used in the proof of the well-foundedness of the lexical coupling.

# Chapter 5

# Implementing Relations using Repetitions

In this chapter we indicate how a specification, i.e. a relation, can be implemented by a program. We call a relation an implementation of a specification if it is, first, implementable and, second, contained in the specification. Of course we now have to make the notion "implementable" somewhat precise. However, we do not make it too precise because that would not really help.

Every programming language, be it imperative or functional, has a number of basic statements or functions. By basic statements we mean statements defined without recursion. For an imperative language, for instance, the assignments are basic. Now we make the (sensible) assumption that programmers know the input-output relations of the basic statements of their favourite language. Therefore we refrain from giving the relational semantics of the basic statements of programming languages (that would be unnecessarily precise). We can say that the input-output relations of basic statements are implementable, because we also assume that a programmer is capable of recognising a relation as the input-output relation of a basic statement.

Of course, there are more implementable relations than just the input-output relations of basic statements. For instance, the relational composition of two implementable relations is implementable: by the composition (in the programming language) of the two implementations.

Every programming language has an alternative construct (if-statement). Therefore, the relation

$$\mathsf{R} \circ \mathsf{B} \cup \mathsf{S} \circ {\sim}\mathsf{B}$$

is implementable if $\mathsf{R}$ and $\mathsf{S}$ are implementable and the characteristic predicate of the monotype $\mathsf{B}$ is expressible in the programming language. Again, we assume that a programmer is capable of recognising such monotypes and knows how to compose the implementations into an $\mathsf{if}$ program. Relations of the form $\mathsf{R} \circ \mathsf{B}$ are in general not implementable if $\mathsf{B}$ differs from I. This is because the programs of a programming language can accept all inputs (are total). However such partial relations can be used to obtain an implementable relation by means of the relation union. We consider a program of the form

$$\mathsf{R} \circ \mathsf{B} \cup \mathsf{S} \circ \mathsf{A}$$

implementable if $\mathsf{B} \cup \mathsf{A} = \mathrm{I}$ (and of course $\mathsf{R}$ and $\mathsf{S}$ are implementable).

Finally, every programming language provides the possibility to write recursive procedures. In this chapter we restrict our attention to procedures with the recursive pattern of the repetition. It is not trivial to define what it means for a relation to be implementable by a recursive procedure. First, it is not obvious what the input-output relation of such a procedure is. Furthermore, for recursive procedures we need to worry about termination. The main concern in this chapter is to make clear what is meant by saying that a relation is implementable by a repetition. The input-output relations of arbitrary recursive programs and their termination conditions are discussed in chapters 7 and 8.

## 5.1 The input-output relation of the repetition

In the guarded command language a repetition is written as $\mathsf{do}\ \mathsf{b} \to \mathsf{s}\ \mathsf{od}$, where $\mathsf{b}$ is a predicate on the statespace and $\mathsf{s}$ is a program. In a functional language it is written as the equation $\mathrm{do}.\mathsf{b}.\mathsf{s}.\mathsf{x} = \mathsf{if}\ \neg(\mathsf{b}.\mathsf{x}) \to \mathsf{x} \,[]\, \mathsf{b}.\mathsf{x} \to \mathrm{do}.\mathsf{b}.\mathsf{s}.(\mathsf{s}.\mathsf{x})\ \mathsf{fi}$. Such a program can be regarded as an implementation of a relation $\mathsf{X}$ if the input-output relation associated with the repetition is contained in $\mathsf{X}$. What then is this input-output relation? To answer this question observe that, if we write do for the repetition, it is the case that do satisfies the following equation.

$$\mathrm{do} \quad = \quad \mathsf{if}\ \neg\mathsf{b} \to \mathsf{skip} \quad [] \quad \mathsf{b} \to \mathsf{s};\mathrm{do}\ \mathsf{fi} \tag{5.1}$$

Translating this to relations —writing $\mathsf{B}$ for the monotype corresponding to predicate $\mathsf{b}$, and $\mathsf{S}$ and DO for the input-output relations of programs $\mathsf{s}$ and do, respectively— we get:

$$\mathrm{DO} \;=\; \sim\!\mathsf{B} \;\cup\; \mathrm{DO} \circ \mathsf{S} \circ \mathsf{B} \tag{5.2}$$

It is even easier to see that this expression is correct if it is compared with the functional form of the repetition that was mentioned above. Note that in expression (5.2) we use the convention that input is from the right as is standard for functional programs.

It is common practice to define the input-output relation of the repetition as the least relation DO satisfying equation (5.2). Of course this raises the question why this solution is the "right" one. A discussion of this question is postponed and we follow the common practice for the moment. However, it will turn out that it does not really matter which solution of (5.2) is chosen.

The least solution of equation (5.2) is $\sim\!\mathsf{B} \circ (\mathsf{S} \circ \mathsf{B})^*$, so we define (temporarily):

**Definition 5.3 (Temporary!)** The input-output relation of program $\mathsf{do}\ \mathsf{b} \to \mathsf{s}\ \mathsf{od}$ is $\sim\!\mathsf{B} \circ (\mathsf{S} \circ \mathsf{B})^*$, where $\mathsf{B}$ is the monotype corresponding to predicate $\mathsf{b}$ and $\mathsf{S}$ the input-output relation of program $\mathsf{s}$.

□

A procedure to implement a relation $\mathsf{X}$ by a repetition is as follows. First we construct a monotype $\mathsf{B}$ and a relation $\mathsf{T}$ such that $\sim\mathsf{B} \cup \mathsf{X}\circ\mathsf{T}\circ\mathsf{B} \subseteq \mathsf{X}$, making sure that the characteristic predicate $\mathsf{b}$ of monotype $\mathsf{B}$ is expressible in the target programming language. Next, relation $\mathsf{T}$ is implemented by program $\mathsf{s}$, i.e. we construct a program $\mathsf{s}$ such that its input-output relation, $\mathsf{S}$ say, satisfies $\mathsf{S}\subseteq\mathsf{T}$. It then easily follows that the input-output relation of program $\mathsf{do\ b} \rightarrow \mathsf{s\ od}$ is contained in relation $\mathsf{X}$.

But there is something more to be said. This has to do with termination. Even when adopting the view that the input-output relation of the repetition $\mathsf{do\ b} \rightarrow \mathsf{s\ od}$ is the least solution of (5.2), we can not be sure that we obtain a "good" implementation of $\mathsf{X}$. To see this consider the following variation of (5.2).

$$\mathrm{DO1} \;=\; \sim\mathsf{B} \;\cup\; \mathrm{DO1}\circ(\mathsf{S}\cup\mathrm{I})\circ\mathsf{B} \tag{5.4}$$

It is not difficult to see that the least solution of equation (5.2) coincides with the least solution of equation (5.4). Let $\mathsf{Y}$ be the least solution of (5.4). Then we have:

$$\begin{array}{ll}
 & \mathsf{Y} \\
= & \quad \{ \ \mathsf{Y} \text{ solves } (5.4) \ \} \\
 & \sim\mathsf{B} \;\cup\; \mathsf{Y}\circ(\mathsf{S}\cup\mathrm{I})\circ\mathsf{B} \\
\supseteq & \quad \{ \ \mathsf{S}\cup\mathrm{I}\supseteq\mathsf{S} \ \} \\
 & \sim\mathsf{B} \;\cup\; \mathsf{Y}\circ\mathsf{S}\circ\mathsf{B}
\end{array}$$

So, by Knaster-Tarski, the least solution $\sim\mathsf{B}\circ(\mathsf{S}\circ\mathsf{B})^*$ of (5.2) is contained in $\mathsf{Y}$. The other inclusion also holds. We can even show that *any* solution $\mathsf{Z}$ of (5.2) solves (5.4) and is therefore contained in the least solution:

$$\begin{array}{ll}
 & \sim\mathsf{B} \;\cup\; \mathsf{Z}\circ(\mathsf{S}\cup\mathrm{I})\circ\mathsf{B} \\
= & \quad \{ \ \text{distributivity, twice; I unit of } \circ \ \} \\
 & \sim\mathsf{B} \;\cup\; \mathsf{Z}\circ\mathsf{S}\circ\mathsf{B} \;\cup\; \mathsf{Z}\circ\mathsf{B} \\
= & \quad \{ \ \mathsf{Z} \text{ solves } (5.2) \ \} \\
 & \mathsf{Z}\cup\mathsf{Z}\circ\mathsf{B} \\
= & \quad \{ \ \mathsf{B}\subseteq\mathrm{I} \text{ therefore } \mathsf{Z}\circ\mathsf{B}\subseteq\mathsf{Z} \ \} \\
 & \mathsf{Z}
\end{array}$$

It follows that, if it is the case that $\mathsf{do\ b} \rightarrow \mathsf{s\ od}$ implements $\mathsf{X}$ then it is also the case that $\mathsf{do\ b} \rightarrow (\mathsf{s} \,[]\, \mathsf{skip})\ \mathsf{od}$ implements $\mathsf{X}$, where $[]$ denotes nondeterministic choice. However, from a computational point of view, this last program has a nasty property: it need not terminate. When started in a state satisfying $\mathsf{b}$ it is possible that in the body the alternative $\mathsf{skip}$ is always chosen. The problem is, of course, that the relation $(\mathsf{S}\cup\mathrm{I})\circ\mathsf{B}$ is not well-founded.

From this discussion it is clear that there is more to equations in relations than their least solution. Equations (5.2) and (5.4) can not be distinguished on their least solutions only. Yet a direct implementation of the second one may lead to a non-terminating program, even if a direct implementation of the first is guaranteed to terminate. Therefore, from a computational point of view they are quite different. To be able to distinguish them, we assign a certain monotype to the equation. This monotype should be defined in such a way that it corresponds to the set of states where the implementation can be started safely, i.e. where the implementation is guaranteed to terminate.

The first requirement on such a set of safe states $A$ is that when the implementation is started in it, the "recursive call" is certainly terminating, that is, is started in the safe set also. This translates to:

$$\mathrm{DO} \circ A \;\subseteq\; {\sim}B \;\cup\; \mathrm{DO} \circ A \circ S \circ B. \tag{5.5}$$

This will be the case if $A$ is contained in $(S \circ B)\backslash A$, the "weakest precondition" of $S \circ B$ in $A$, as is shown in the following calculation.

$$\begin{array}{ll}
 & {\sim}B \;\cup\; \mathrm{DO} \circ A \circ S \circ B \\
\supseteq & \qquad \{\ (S \circ B)\backslash A \subseteq \mathrm{I};\ \text{monotype factors (2.43)}\ \} \\
 & {\sim}B \circ (S \circ B)\backslash A \;\cup\; \mathrm{DO} \circ S \circ B \circ (S \circ B)\backslash A \\
= & \qquad \{\ \text{distributivity}\ \} \\
 & ({\sim}B \;\cup\; \mathrm{DO} \circ S \circ B) \circ (S \circ B)\backslash A \\
= & \qquad \{\ \mathrm{DO}\ \text{solves (5.2)}\ \} \\
 & \mathrm{DO} \circ (S \circ B)\backslash A \\
\supseteq & \qquad \{\ \text{assume: } A \subseteq (S \circ B)\backslash A\ \} \\
 & \mathrm{DO} \circ A
\end{array}$$

On the other hand, if we forget about the last step, the above calculation reveals that we also have:

$$\mathrm{DO} \circ (S \circ B)\backslash A \;\subseteq\; {\sim}B \;\cup\; \mathrm{DO} \circ A \circ S \circ B. \tag{5.6}$$

This is to be interpreted as: if the implementation is started in $(S \circ B)\backslash A$, the recursive call is started in the safe set $A$. Therefore we can "safely" assume that $(S \circ B)\backslash A$ is contained in $A$. Combining the two assumptions, we get that the safe set $A$ should satisfy $A = (S \circ B)\backslash A$. Now we take the pessimistic (but realistic) view that what can go wrong will, i.e. the set of non-safe states is large, and therefore the set of safe states is small. So we define: the safe monotype is the *least* monotype $A$ such that $A = (S \circ B)\backslash A$ . That is, $A = \mu((S \circ B)\backslash)$.

We have seen before (theorem 3.25) that the relation $S \circ B$ restricted to the monotype $\mu((S \circ B)\backslash)$ is well-founded. This means that for any $x \in \mu((S \circ B)\backslash)$ there does not exist an infinite $S \circ B$-chain starting from $x$. Therefore, the fact that the implementation has to be started in this monotype complies with the usual statement that the repetition is guaranteed to terminate when started in this set. This set indeed corresponds to the predicate wp.(do b $\rightarrow$ s od).true.

Now we return to definition 5.3. It is to be adapted to cater for the fact that the repetition has to be started in states from the monotype $\mu((\mathsf{S}\circ\mathsf{B})\backslash)$. We are not primarily interested in the input-output relation of the repetition itself. The important thing to define is the restriction of this relation to the monotype $\mu((\mathsf{S}\circ\mathsf{B})\backslash)$. That relation then is $\sim\mathsf{B}\circ(\mathsf{S}\circ\mathsf{B})^*\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash)$, based on the assumption that the input-output relation of the repetition is the least solution of equation (5.2). But now we are in a position to drop that assumption. It is only needed that the input-output relation of the repetition is *some* solution of equation (5.2). This is because of the following theorem.

**Theorem 5.7** Consider the following equation in $\mathsf{X}$

$$\mathsf{X} = \sim\mathsf{B} \ \cup\ \mathsf{X}\circ\mathsf{S}\circ\mathsf{B} \tag{5.8}$$

If $\mathsf{Y}$ is the least, and DO an arbitrary solution of this equation then the following holds.

$$\mathrm{DO}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) = \mathsf{Y}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash)$$

**Proof**

$$
\begin{array}{cl}
 & \mathrm{DO}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) = \mathsf{Y}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) \\
\equiv & \quad \{\ \mathsf{Y} \text{ is the least solution, so } \mathsf{Y}\subseteq\mathrm{DO}\ \} \\
 & \mathrm{DO}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) \subseteq \mathsf{Y}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) \\
\equiv & \quad \{\ \Rightarrow:\ \mu((\mathsf{S}\circ\mathsf{B})\backslash)\subseteq \mathrm{I} \\
 & \quad\ \ \Leftarrow:\ \text{composition is monotone and monotypes are transitive}\ \} \\
 & \mathrm{DO}\circ\mu((\mathsf{S}\circ\mathsf{B})\backslash) \subseteq \mathsf{Y} \\
\Leftarrow & \quad \{\ \mu\text{-fusion; } (\mathrm{DO}\circ) \text{ has an upper adjoint. For all } \mathsf{A}\ \} \\
 & \mathrm{DO}\circ(\mathsf{S}\circ\mathsf{B})\backslash\mathsf{A} \ \subseteq\ \sim\mathsf{B}\ \cup\ (\mathrm{DO}\circ\mathsf{A})\circ\mathsf{S}\circ\mathsf{B}. \\
\equiv & \quad \{\ \text{see (5.6)}\ \} \\
 & \text{true}
\end{array}
$$

□

It follows from this theorem that there is a close connection between termination and unicity of solutions. If $\mu((\mathsf{S}\circ\mathsf{B})\backslash)$ equals the identity relation I, so if termination is always guaranteed, equation (5.8) has a unique solution. Or, by the contrapositive, an equation with more than one solution can lead to a non-terminating implementation. This explains why it is interesting to study equations in programs with a unique solution, as is done in for instance the squiggol literature. There, a large variety of so-called morphisms are studied, those morphisms all arising as unique solutions of equations.

From theorem 5.7 it is also clear that we need only know that the input-output relation of the repetition is *some* solution of equation (5.8) to be able to infer that this relation, when restricted to the safe monotype, is $\sim B \circ (S \circ B)^* \circ \mu((S \circ B)\searrow)$. Of course, if this is all we know, we can not say much about the rest of the input-output relation of the repetition. But that is not much of a problem. When started in states which are not in the safe monotype the repetition might not terminate. As this is bad behaviour, which we want to avoid anyway, we do not care what the input-output relation restricted to the complement of the safe monotype is. Therefore we define the input-output relation of the repetition only partially in the following way.

**Definition 5.9 (Repetition)** The input-output relation of program $\mathsf{do}\ b \rightarrow s\ \mathsf{od}$ is a solution of the equation in $X$:

$$X = \sim B \ \cup \ X \circ S \circ B$$

where $B$ is the monotype corresponding to predicate $b$ and $S$ is the input-output relation of $s$.
□

In the usual approach to relational semantics, non-termination is modelled by adding an extra element $\bot$, pronounced bottom. The fact $\bot\langle R\rangle x$ is to be interpreted as: $R$ need not terminate when started in $x$. In that approach the semantics of the repetition is defined as the least fixed point under a rather complicated partial order on relations, called the Egli-Milner order. The least solution under the normal inclusion order can then be recovered by restricting the Egli-Milner least solution to values different from $\bot$. The safe set is formed by the elements that are not related to $\bot$.

Defining the repetition the way we do has the advantage that the addition of a special element to the universe to express non-termination is not needed. Another advantage is that it is not necessary to introduce the Egli-Milner order. Moreover, if we assume that $\bot$ is not an element of the domain of $S$, we do not loose anything. The Egli-Milner least fixed point can be expressed as $\mathrm{DO} \cup \mathrm{NT}$ (see for instance [84, 38], where DO is the least solution of (5.2) and NT is the relation defined as follows.

$$x\langle \mathrm{NT}\rangle y \ \equiv \ x = \bot \wedge y \notin \mu((S \circ B)\searrow)$$

Of course, to substantiate these claims we *have* to define the Egli-Milner order and consider the effect of adding a bottom element. This will be done in chapter 8.

Defining the input-output relation of the repetition as in definition (5.9) has another nice consequence. It forces the programmer to give a termination proof. Proving that a relation $X$ satisfies equation (5.8) is not enough to obtain an implementation using a repetition, this being only partial correctness. The programmer can not prove in this case that the input-output relation, $Y$ say, of the repetition is contained in $X$, because he can not prove that $Y$ is the least solution of (5.8). What is needed for total correctness is also a proof that the repetition is used in a "context" where its input-output relation $Y$ can be replaced by the least solution $Y \circ \mu((S \circ B)\searrow)$. Stated differently, for total correctness it is needed that the repetition is used in an environment where the precondition $\mu((\mathsf{S}\circ\mathsf{B})\searrow)$ can be relied on.

To illustrate how well it all can work, consider as an example the linear search program. Its input-output relation is a solution of the equation in $\mathsf{X}$:

$$\mathsf{X} \;=\; \mathsf{A} \,\cup\, \mathsf{X}\circ\mathsf{succ}\circ{\sim}\mathsf{A} \tag{5.10}$$

where $\mathsf{succ}$ is the successor function on the natural numbers. The safe monotype of this equation is $\mu((\mathsf{succ}\circ{\sim}\mathsf{A})\searrow)$ which is the same (because $\mathsf{succ}$ is a total function see (2.46)) as $(\mathsf{A}\circ\mathsf{succ}^*)>$. The reflexive and transitive closure of the successor function is of course the relation $\geq$. Therefore $x\in\mu((\mathsf{succ}\circ{\sim}\mathsf{A})\searrow)$ is the same as $x\in(\mathsf{A}\circ(\geq))>$, i.e. there exists an element $y$ satisfying $\mathsf{A}$ such that $y\geq x$. And this is exactly the precondition under which we expect the obvious implementation $\mathsf{do}\ \neg(a.x) \to x := x{+}1\ \mathsf{od}$ of the linear search to be correct.

## 5.2 Programming with invariants

### 5.2.1 Introduction

A, by now standard, way of developing programs is the technique of using invariants. In this method a specification $\mathsf{X}$ is implemented by a program of the form $r\ ;\ \mathsf{do}\ b \to s\ \mathsf{od}\ ;\ t$. This program is constructed by first choosing a suitable invariant and a guard $b$ (both predicates on the state space).

The invariant has to be initialised by program $r$, i.e. when started in the precondition of $\mathsf{X}$, program $r$ has to terminate in a state satisfying the invariant.

The requirement on program $s$ is that it maintains the invariant: when started in a state satisfying the guard and the invariant, program $s$ terminates in a state which satisfies the invariant. Another requirement, called the termination condition, boils down to the fact that the input-output relation of program $s$, restricted to states satisfying the invariant and the guard, is well-founded.

The requirement on program $t$ is that, when started in a state satisfying $\neg b$ and the invariant, it terminates in a state satisfying the postcondition of specification $\mathsf{X}$.

The requirements on invariant, guard and programs $r$, $s$ and $t$ are of great help during the process of designing the implementation of $\mathsf{X}$ and this method of program development has proved to be one of the most successful around. Many programs have been developed using it in a beautiful way and a lot of heuristics have been developed to guide the design process. However, it is only applicable to the design of sequential programs. It is possible to define functions with the recursion pattern of the do-loop in a functional language. But it is not possible to develop such a program in the style sketched above using a calculus of functions. This is due to the fact that the invariant is not a function: it is a relation.

Because the invariant is a relation it should be possible to develop repetitions using a relational calculus. In this section it is shown how this can be done. It will turn out that it is possible to formulate requirements such that $\mathsf{T}\circ\mathsf{Y}\circ\mathsf{R}\subseteq\mathsf{X}$, where the only restriction on relation $\mathsf{Y}$ is that it solves $\mathsf{Y} = \sim\!\mathsf{A} \cup \mathsf{Y} \circ \mathsf{S} \circ \mathsf{A}$ . In other words $\mathsf{T} \circ \mathsf{Y} \circ \mathsf{R}$ refines specification $\mathsf{X}$ where $\mathsf{Y}$ can be implemented by a repetition, relations $\mathsf{T}$ and $\mathsf{R}$ then taking the role of finalisation and initialisation, respectively. Furthermore, it is possible to formulate these requirements in such a way that the large body of knowledge developed for programming in the imperative style can be used. This is illustrated by several simple examples.

### 5.2.2 Relational programming with invariants

Suppose we have to implement a relation $\mathsf{X}$. The first step is to choose a relation $\mathsf{P}$, the invariant, a relation $\mathsf{T}$, the finalisation, and a monotype $\mathsf{A}$, the guard, such that

$$\mathsf{T} \circ \sim\!\mathsf{A} \circ \mathsf{P} \subseteq \mathsf{X} \quad . \tag{5.11}$$

In the construction of these relations we may use the well-known techniques such as replacing a constant by a variable and deleting a conjunct. Usually the left domain of the invariant is a product type with more components than the left domain of the specification. Therefore, the finalisation is often just a projection function.

The next step is to find a relation $\mathsf{R}$, the initialisation, with the property

$$\mathsf{R} \subseteq \mathsf{P} \quad . \tag{5.12}$$

A design consideration in the construction of the invariant can be the existence of a simple initialisation. Finally, and this is the most difficult part, we have to construct a relation $\mathsf{S}$, called the body, which has to satisfy the requirement

$$\mathsf{S} \circ \mathsf{A} \circ \mathsf{P} \subseteq \mathsf{P} \quad . \tag{5.13}$$

In the construction of the body it may turn out that the invariant has to be strengthened, for instance by adding a new component to its left domain or by requiring additional relations between these components. Requirement (5.13) is what is called the invariance of the invariant under the body. In practice it is more convenient to formulate it using points: for all $x$, $y$ and $z$

$$x\langle \mathsf{S}\rangle y \wedge y\langle \mathsf{P}\rangle z \wedge y{\in}\mathsf{A} \quad \Rightarrow \quad x\langle \mathsf{P}\rangle z \tag{5.14}$$

So, we might construct relation $\mathsf{S}$ in a calculation of the following form.

$$\begin{array}{ll} & x\langle \mathsf{P}\rangle z \\ \Leftarrow & \qquad \{ \quad \text{assume } y\langle \mathsf{P}\rangle z \wedge y{\in}\mathsf{A} \quad \} \\ & x\langle \mathsf{S}\rangle y \end{array}$$

If we are heading for a functional body, expression (5.13) can be rewritten as $\mathsf{A} \circ \mathsf{P} \subseteq \mathsf{S}^{\cup} \circ \mathsf{P}$ or, formulated using points, $x{\in}\mathsf{A} \wedge x\langle \mathsf{P}\rangle y \Rightarrow \mathsf{S}.x\langle \mathsf{P}\rangle y$. In this case we can try to construct function $\mathsf{S}$ in a calculation of the form

$$
\begin{array}{ll}
 & x\langle P\rangle y \\
\Rightarrow & \quad \{ \text{ assume } x{\in}A \quad \} \\
 & S.x\langle P\rangle y
\end{array}
$$

An additional requirement on body, guard and invariant corresponds to the termination proof.

$$S \circ A \circ P_{<} \text{ is well-founded} \tag{5.15}$$

We can try to prove this using the theory of chapter 3, for instance by proving the existence of a function $f$ to the natural numbers with the property

$$S \circ A \circ P_{<} \subseteq f^{\cup} \circ < \circ f \tag{5.16}$$

Or formulated pointwise: $x\langle S\rangle y \wedge y{\in}A \wedge y \in P_{<} \Rightarrow f.x < f.y$. This is the technique that is usually used in the imperative style for the correctness proof of repetitions.

### 5.2.3 The repetition theorem

In this section we formulate and prove the theorem which shows that relations $R$, $S$, $T$ and $A$ which satisfy the requirements (5.11), (5.12) and (5.13) of the previous section can indeed be used to implement the relation $X$. As a first stepping stone we have the following theorem.

**Theorem 5.17 (The invariance theorem)** If relations $P$, $R$, $S$, $T$, $X$ and monotype $A$ satisfy the conditions (5.11), (5.12) and (5.13) we have

$$T \circ {\sim}A \circ (S \circ A)^{*} \circ R \subseteq X \quad .$$

**Proof**

$$
\begin{array}{ll}
 & T \circ {\sim}A \circ (S \circ A)^{*} \circ R \subseteq X \\
\Leftarrow & \quad \{ \text{ assumption (5.11) } \} \\
 & T \circ {\sim}A \circ (S \circ A)^{*} \circ R \subseteq T \circ {\sim}A \circ P \\
\Leftarrow & \quad \{ \text{ monotonicity of composition } \} \\
 & (S \circ A)^{*} \circ R \subseteq P \\
\Leftarrow & \quad \{ \text{ reflexive transitive closure:} \\
 & \qquad U^{*} \circ V \text{ is the least solution of the equation in } Y \\
 & \qquad V \cup U {\circ} Y \subseteq Y \quad \} \\
 & R \cup S \circ A \circ P \subseteq P \\
\equiv & \quad \{ \text{ assumptions (5.12) and (5.13) } \} \\
 & \text{true}
\end{array}
$$

□

This theorem suggests how to implement $X$ if we are able to implement relations $R$, $S$, $T$ and $A$. Abbreviating $\sim A \circ (S \circ A)^*$ to $Y$, it is clear that $Y$ is the least solution of the equation in $Z$: $Z = \sim A \cup Z \circ S \circ A$. If we are now able to show that $Y$ can be replaced by an arbitrary such $Z$, it follows from definition 5.9 that $Y$ can be implemented by a repetition. According to theorem 5.7, relation $Y$ can indeed be replaced by an arbitrary relation $Z$ satisfying $Z = \sim A \cup Z \circ S \circ A$ if we can prove that $T \circ Y \circ R$ equals $T \circ Y \circ \mu((S \circ A)\searrow) \circ R$. This is where requirement (5.15) comes in. Now we show how that condition can be derived.

$$
\begin{array}{ll}
 & T \circ Y \circ R = T \circ Y \circ \mu((S \circ A)\searrow) \circ R \\
\Leftarrow & \quad \{ \text{ monotonicity of composition } \} \\
 & R = \mu((S \circ A)\searrow) \circ R \\
\equiv & \quad \{ \text{ domains } \} \\
 & R_{<} \subseteq \mu((S \circ A)\searrow) \\
\Leftarrow & \quad \{ \text{ (5.12), monotonicty of the domain operator } \} \\
 & P_{<} \subseteq \mu((S \circ A)\searrow) \\
\equiv & \quad \{ \text{ assume: } X \text{ is well-founded, i.e. } \mu(X\searrow) = I \} \\
 & P_{<} \circ \mu(X\searrow) \subseteq \mu((S \circ A)\searrow) \\
\Leftarrow & \quad \{ \mu\text{-fusion; } (P_{<} \circ) \text{ has an upper adjoint; for all } B \} \\
 & P_{<} \circ X\searrow B \subseteq (S \circ A)\searrow(P_{<} \circ B) \\
\equiv & \quad \{ \text{ monotype factors } \} \\
 & (S \circ A \circ P_{<} \circ X\searrow B)_{<} \subseteq P_{<} \circ B \\
\Leftarrow & \quad \{ \circ \text{ is intersection on monotypes; } X\searrow B \subseteq I \} \\
 & (S \circ A \circ P_{<})_{<} \subseteq P_{<} \\
 & \wedge\ (S \circ A \circ P_{<} \circ X\searrow B)_{<} \subseteq B \\
\Leftarrow & \quad \{ \text{ domains; assume: } S \circ A \circ P_{<} \subseteq X \} \\
 & (S \circ A \circ P)_{<} \subseteq P_{<} \\
 & \wedge\ (X \circ X\searrow B)_{<} \subseteq B \\
\equiv & \quad \{ \text{ (5.13), monotonicity of the domain operator;} \\
 & \quad\quad \text{factor cancellation } \} \\
 & \text{true}
\end{array}
$$

So, provided that $S \circ A \circ P_{<}$ is well-founded we may replace the relation $\sim A \circ (S \circ A)^*$ in the statement of theorem 5.17 by an arbitrary relation $Z$ satisfying $Z = \sim A \cup Z \circ S \circ A$ and we have proved the next theorem.

**Theorem 5.18 (The repetition theorem)** Let relations $\mathsf{R}$, $\mathsf{S}$, $\mathsf{T}$, $\mathsf{P}$, $\mathsf{X}$ and $\mathsf{A}$ satisfy the conditions (5.11), (5.12), (5.13) and (5.15), and let relation $\mathsf{Z}$ satisfy $\mathsf{Z} = {\sim}\mathsf{A} \cup \mathsf{Z} \circ \mathsf{S} \circ \mathsf{A}$. Then we have

$$\mathsf{T} \circ \mathsf{Z} \circ \mathsf{R} \subseteq \mathsf{X} \quad .$$

□

### 5.2.4 Examples

In this section we provide some examples of how programs can be constructed by using the conditions of theorem 5.18 as the guiding restrictions. It is not claimed that this is the best way to construct programs or that no improvements are possible. The example program constructions are all carried out on the point level and can therefore not be considered to make a case for the use of point-free relation calculus in programming. It might very well be possible to develop the relation calculus further so as to make it possible to derive the programs of this section in a point-free way.

Although not point-free, the derivations can be seen as examples of pointwise relational programming. What we want to do is, first, to show that it is possible to derive programs this way and, second, that the calculations that have to be done boil down to those that are needed for the well-known discipline of developing programs in the imperative style. As such the examples can be seen as justification of the claim that this well-known discipline is in fact relational programming.

Because these are our only goals, we do not go all the way to an implementation in a concrete programming language but stop at the moment that the construction of such an implementation becomes straightforward. Also we have to admit that although the specifications are extremely simple, the calculations that lead to the implementation are rather complicated. However, it is hoped that the examples give evidence that the relation calculus can be developed further in such a way that any program that can be developed in the imperative style could also be developed by using relational methods.

**Tail recursion: the factorial**

As a first example of what the relational style of programming should look like, we develop an implementation of the (obligatory) factorial function.

$$a\langle \mathsf{X}\rangle n \quad \equiv \quad a = n! \wedge n \geq 0 \tag{5.19}$$

We construct a (tail) invariant by replacing the (invisible) constant 1 by the expression $k!$ and adding a conjunct to restrict the range of $k$.

$$(a,k)\langle \mathsf{P}\rangle n \quad \equiv \quad a \times k! = n! \ \wedge \ 0 \leq k \tag{5.20}$$

It is now obvious that we have to define $(a,k) \notin \mathsf{A}$ as: $k = 0$ . The finalisation $\mathsf{T}$ is the projection on the first coordinate: $b\langle \mathsf{T}\rangle (a,k) \quad \equiv \quad a = b$ or $\mathsf{T}.(a,k) = a$. The initialisation is easy this way : $(a,k)\langle \mathsf{R}\rangle n \ \equiv \ a = 1 \wedge k = n$, i.e. $\mathsf{R}.n = (1,n)$. To construct the body, we calculate.

$$
\begin{array}{ll}
 & (a,k)\langle A \circ P \rangle n \\
\equiv & \{ \text{ definitions } \} \\
 & a{\times}k! = n! \wedge 0 \le k \wedge k \ne 0 \\
\Rightarrow & \{ \text{ factorial function and arithmetic } \} \\
 & a \times k \times (k-1)! = n! \ \wedge \ 0 \le k-1 \\
\equiv & \{ \text{ definition } \} \\
 & (a{\times}k\,, k-1)\langle P \rangle n
\end{array}
$$

Therefore, we can define $S.(a,k) = (a{\times}k\,, k{-}1)$. All functions involved are implementable, so the only thing left to check is the well-foundedness of $S \circ A \circ P_{<}$. In this case that is easy because $S \circ A \circ P_{<} \subseteq \top\!\!\top \times \mathsf{pred}$. Recall from chapter 3 that for well-founded $R$ the relation $\top\!\!\top \times R$ is well-founded also (see the section on the lexical coupling). Observe, furthermore, that the $\mathsf{pred}$ function, i.e. the inverse of the successor function on natural numbers, is the canonical example of a well-founded relation.

### Searching for the maximum of a function

The problem is to find a value in an interval of natural numbers where a given function takes its maximum. This problem is treated in somewhat greater detail. This is only done to give a good idea of the programming method.

$$m\langle X \rangle n \quad \equiv \quad 0 \le m \le n \ \wedge \ f.m = \mathrm{MAX}(i:\ 0 \le i \le n:\ f.i) \tag{5.21}$$

To make the calculations a bit more compact an abbreviation for the quantified expression is introduced.

$$M.n = \mathrm{MAX}(i:\ 0 \le i \le n:\ f.i) \tag{5.22}$$

An invariant is obtained in the standard way by replacing the constant $n$ by the variable $k$, and adding the relevant bound to this variable. In this context a constant is a value from the right domain of the specification, while a variable is a value from its left domain. So the invariant becomes:

$$(m,k)\langle P \rangle n \quad \equiv \quad 0 \le m \le k \le n \ \wedge \ f.m = M.k \tag{5.23}$$

Note that the righthand side of this expression is exactly the invariant that would be used in a development of an imperative program. The extra information that expression (5.23) gives, is what is input, or constant, (namely $n$) and output, or variable, ( $m$ and $k$).

Now $k = n \ \wedge \ (m,k)\langle P \rangle n$ implies $m\langle X \rangle n$. However this can not be expressed as $(m,k)\langle {\sim}A \circ P \rangle n$ because to do that we need $(m,k) \notin A$ equivales $k = n$. And this does not work because the value of $n$ is not available in the pair $(m,k)$. Therefore, we have to add a new variable that has this value.

$$(m,k,l)\langle P_0 \rangle n \quad \equiv \quad 0 \le m \le k \le n \wedge f.m = M.k \wedge l = n \tag{5.24}$$

If we now define as guard $(m,k,l){\in}A \;\equiv\; k \neq l$ we do have that $(m,k,l)\langle{\sim}A \circ P_0\rangle n$ implies $m\langle X\rangle n$. What the finalisation should be is then also clear: a projection on the first coordinate. It is also not necessary to say much about the initialisation. A quick calculation using the one point rule shows that we can take for $m$ and $k$ the value $0$. For $l$ we have to take $n$. Therefore the definitions of finalisation and initialisation are as follows.

$$R.(m,k,l) = m \quad , \quad (m,k,l)\langle T\rangle n \;\equiv\; m = k = n \wedge l = n \wedge 0 \leq n \tag{5.25}$$

The final part to construct is the body. We assume $(m_0,k_0,l_0)(A \circ P_0)n$, i.e.

$$k_0 \neq l_0 \wedge 0 \leq m_0 \leq k_0 \leq n \;\wedge\; f.m_0 = M.k_0 \wedge l_0 = n \tag{5.26}$$

We have to construct is a relation $S$ such that $(m,k,l)\langle S\rangle(m_0,k_0,l_0)$ implies $(m,k,l)(P_0)n$. For then it is the case that $S \circ A \circ P_0 \subseteq P_0$. Of course we have to make sure that $S \circ A \circ P_0{<}$ is well-founded. In the next calculation the pointwise expression of this relation is calculated.

$$\begin{array}{ll}
 & (m,k,l)\langle S \circ A \circ P_0{<}\rangle(m_0,k_0,l_0) \\
\equiv & \qquad \{ \text{ relational composition; monotypes } \} \\
 & (m,k,l)\langle S\rangle(m_0,k_0,l_0) \;\wedge\; (m_0,k_0,l_0){\in}A \;\wedge\; (m_0,k_0,l_0) \in P_0{<} \\
\equiv & \qquad \{ \text{ definition of } A\text{; domains } \} \\
 & (m,k,l)\langle S\rangle(m_0,k_0,l_0) \;\wedge\; k_0 \neq l_0 \;\wedge\; \exists(n :: \; (m_0,k_0,l_0)(P_0)n) \\
\equiv & \qquad \{ \text{ definition (5.23) and one-point rule: } l_0 = n \; \} \\
 & (m,k,l)\langle S\rangle(m_0,k_0,l_0) \;\wedge\; k_0 \neq l_0 \;\wedge\; 0 \leq m_0 \leq k_0 \leq l_0 \;\wedge\; f.m_0 = M.k_0
\end{array}$$

From this last expression it can be seen that the component $k_0$ of the right domain of $S \circ A \circ P_0{<}$ is bounded by the component $l_0$. Therefore it is a good idea to make sure that $S$ does not change this third component, i.e $l = l_0$ , and that the second component is increased by $1$, thus $k = k_0{+}1$. If we then define a bound function by $f.(m,k,l) = l - k$ for $(m,k,l) \in P_0{<}$ it is straightforward to show that $f$ is a function into the natural numbers and, furthermore that:

$$S \circ A \circ P_0{<} \subseteq f^{\cup} \circ (<) \circ f$$

Of course it is not necessary to calculate the expression for $(m,k,l)\langle S \circ A \circ P_0{<}\rangle(m_0,k_0,l_0)$. The argument above is just as easily seen to be correct by considering expression (5.26). So now well-foundedness is proved. Now we construct the rest of relation $S$.

$$\begin{array}{ll}
 & (m,k,l)(P_0)n \\
\equiv & \qquad \{ \text{ definition (5.24) } \} \\
 & 0 \leq m \leq k \leq n \;\wedge\; f.m = M.k \wedge l = n \\
\equiv & \qquad \{ \text{ assume: } k = k_0{+}1,\; l_0 = l \; \}
\end{array}$$

$$
\begin{array}{ll}
& 0 \le m \le k_0{+}1 \le n \;\wedge\; f.m = M.(k_0{+}1) \;\wedge\; l_0 = n \\
\equiv & \quad \{ \;\; (5.26)\text{: } k_0{+}1 \le n \text{ and } l_0 = n \text{ ; quantors } \;\; \} \\
& 0 \le m \le k_0{+}1 \;\;\wedge\;\; f.m \;=\; M.k_0 \max f.(k_0{+}1) \\
\equiv & \quad \{ \;\; (5.26)\text{: } f.m_0 = M.k_0 \;\; \} \\
& 0 \le m \le k_0{+}1 \;\;\wedge\;\; f.m = f.m_0 \max f.(k_0{+}1) \\
\equiv & \quad \{ \;\; \text{assume: } f.m_0 \le f.(k_0{+}1) \;\; \} \\
& 0 \le m \le k_0{+}1 \;\;\wedge\;\; f.m = f.(k_0{+}1) \\
\equiv & \quad \{ \;\; \text{assume: } m = k_0{+}1 \;\; \} \\
& 0 \le k_0{+}1 \\
\equiv & \quad \{ \;\; (5.26)\text{: } 0 \le k_0 \;\; \} \\
& \text{true}
\end{array}
$$

Collecting the requirements on relation $S$ :

$$(m,k,l)\langle S\rangle(m_0,k_0,l_0) \quad\equiv\quad m = k_0{+}1 \;\wedge\; k = k_0{+}1 \;\wedge\; l_0 = l \;\wedge\; f.m_0 \le f.(k_0{+}1)$$

This relation is not total. It is only defined when $f.m_0 \le f.(k_0{+}1)$, and is therefore not implementable. However, by a similar calculation, we also have that relation $U$, defined as in the next formula, satisfies the requirements for the body.

$$(m,k,l)\langle U\rangle(m_0,k_0,l_0) \quad\equiv\quad m = m_0 \;\wedge\; k = k_0{+}1 \;\wedge\; l_0 = l \;\wedge\; f.(k_0{+}1) \le f.m_0$$

So now we have that both $S \circ A \circ P_0 \subseteq P_0$ and $U \circ A \circ P_0 \subseteq P_0$ hold, and therefore, by simple relation calculus, $(S \cup U) \circ A \circ P_0 \subseteq P_0$. This suggests that we try to implement the body by an if-program. If we define monotypes $B$ and $C$ as follows, we can rewrite $(S \cup U)$ in a suitable form.

$$(m_0,k_0,l_0) \in B \;\equiv\; f.m_0 \le f.(k_0{+}1) \qquad (m_0,k_0,l_0) \in C \;\equiv\; f.(k_0{+}1) \le f.m_0$$

Using these monotypes, relations $S$ and $U$ are rewritten as $V \circ B$ and $W \circ C$ respectively. But now the union of the monotypes $B$ and $C$ is the identity relation. So the body can indeed be implemented by an if-program. Here relations $V$ and $W$ are functional and defined by:

$$V.(m_0,k_0,l_0) = (k_0{+}1\,, k_0{+}1\,, l_0) \qquad W.(m_0,k_0,l_0) = (m_0\,, k_0{+}1\,, l_0)$$

Note that because monotypes $B$ and $C$ are not disjoint, the body $V \circ B \cup W \circ C$ is not functional. This ends the development: initialisation, guard, body and finalisation are constructed in such a way that the conditions of the repetition theorem (5.18) are fulfilled. Implementation should be a straightforward exercise from this point on.

### Squaring a number

To illustrate the technique of strengthening the invariant we now develop a program that squares a natural number by using additions only. The specification is the following.

$$m\langle X\rangle n \;\equiv\; m = n^2 \wedge 0 \leq n \tag{5.27}$$

We construct the invariant by replacing "constant" $n$ by a "variable" and adding the bounds of this variable. Because we do not know beforehand how many variables we need, we introduce a vector $a$ of them. The components of this vector are called $a_0$, $a_1$ etc. and $a_0$ will take the role of $m$.

$$a\langle P_0\rangle n \;\equiv\; a_0 = a_1^2 \;\wedge\; 0 \leq a_1 \leq a_2 = n \quad . \tag{5.28}$$

Initialisation is straightforward: if $a_0 = a_1 = 0$ and $a_2 = n$, then we have $a\langle P_0\rangle n$, provided that $0 \leq n$ . So we take as initialisation relation $R_0$

$$a\langle R_0\rangle n \;\equiv\; a_0 = a_1 = 0 \wedge a_2 = n \wedge 0 \leq n \quad . \tag{5.29}$$

Now for the finalisation and guard. This is also standard, if $a_1 = a_2$ it is the case that $a(P_0)n$ implies $a_0 = n^2$. So, a suitable finalisation is:

$$T.a = a_0 \quad . \tag{5.30}$$

The guard becomes:

$$a \in A \;\equiv\; a_1 \neq a_2 \quad . \tag{5.31}$$

The most difficult part is the construction of the body. We assume $a\langle P_0\rangle n$ and $a \in A$:

$$a_0 = a_1^2 \;\wedge\; 0 \leq a_1 \;\wedge\; a_1 \leq n \wedge a_2 = n \;\wedge\; a_1 \neq a_2 \quad . \tag{5.32}$$

What we have to construct is a relation $S$ such that $b\langle S\rangle a$ implies the following formula i.e. $b\langle P_1\rangle n$ .

$$b_0 = b_1^2 \;\wedge\; 0 \leq b_1 \;\wedge\; b_1 \leq n \wedge b_2 = n \quad . \tag{5.33}$$

In the construction we have to bear in mind that $S \circ A \circ P_<$ has to be well-founded. This will be the case if the second component of $b$ is greater than the second component of $a$ (which on account of $P_<$ is bounded above by $n$). So we now take the decision that $b\langle S\rangle a$ implies $b_1 = a_1{+}1$. We try to construct the rest of $S$ in the next calculation.

$$
\begin{array}{cl}
 & b_0 = b_1^2 \;\wedge\; 0 \leq b_1 \;\wedge\; b_1 \leq n \wedge b_2 = n \\
\equiv & \qquad \{ \;\text{assumption } b_1 = a_1{+}1 \; \} \\
 & b_0 = (a_1{+}1)^2 \;\wedge\; 0 \leq a_1{+}1 \;\wedge\; a_1{+}1 \leq n \;\wedge\; b_2 = n \\
\equiv & \qquad \{ \;\text{arithmetic} \; \} \\
 & b_0 = a_1^2{+}(2{\times}a_1){+}1 \;\wedge\; 0 \leq a_1 \;\wedge\; a_1 \leq n \;\wedge\; a_1 \neq n \wedge b_2 = n \\
\Leftarrow & \qquad \{ \;\text{assumption (5.32)} \; \} \\
 & b_0 = a_0{+}(2{\times}a_1){+}1 \;\wedge\; b_2 = a_2
\end{array}
$$

From this calculation we see that relation $S$ has to satisfy $b\langle S\rangle a$ implies $b_2 = a_2$. However, because we were not allowed to use multiplication, we can not require $b_0 = a_0+(2\times a_1)+1$. The easy way out is of course to rewrite the multiplication by 2 as an addition, but we want to illustrate the technique of strengthening the invariant. So, we take the complicated way out and add the conjunct $a_3 = (2\times a_1)+1$ to the invariant. Then we can get rid of the multiplication: $b_0 = a_0+a_3$. The price to be paid is first that we have to adapt the initialisation and second that we have to make sure that $b\langle S\rangle a$ implies $b_3 = (2\times b_1)+1$. In the second proof obligation we may use the newly added conjunct $a_3 = (2\times a_1)+1$ of the invariant.

First the new initialisation. Recall (see (5.29)) that $a(R_1)n$ implies $a_1 = 0$. Therefore we have to add $a_3 = (2\times 0)+1 = 1$ as a conjunct to the initialisation. The extra requirement on the body is calculated as follows.

$$
\begin{array}{ll}
 & b_3 = (2\times b_1)+1 \\
\equiv & \qquad \{ \text{ assumption } b_1 = a_1+1 \ \} \\
 & b_3 = (2\times(a_1+1))+1 \\
\equiv & \qquad \{ \text{ arithmetic } \} \\
 & b_3 = (2\times a_1)+2+1 \\
\equiv & \qquad \{ \text{ assumption } a_3 = (2\times a_1)+1 \ \} \\
 & b_3 = a_3+2
\end{array}
$$

Collecting the requirements on relation $S$ we see that we can take as its definition:

$$b\langle S\rangle a \ \equiv \ b_0 = a_0+a_3 \wedge b_1 = a_1+1 \wedge b_2 = a_2 \wedge b_3 = a_3+2 \tag{5.34}$$

For completeness sake: the definition of the initialisation,

$$a(R_1)n \quad \equiv \quad a_0 = a_1 = 0 \ \wedge \ 0\leq n \ \wedge \ a_2 = n \ \wedge \ a_3 = 1 \tag{5.35}$$

and the definition of the guard and finalisation:

$$a\in A \ \equiv \ a_1 \neq a_2 \qquad T.a = a_0 \tag{5.36}$$

Using these definitions and the Repetition Theorem it is now a straightforward exercise to implement the squaring function in either a functional or an imperative programming language. Note that the initialisation relation $R$ is defined on the natural numbers only. Therefore the implementation will only be defined on the natural numbers, i.e. the implementation seems to be partial. But that is not really the case: the implementation is, like the specification, total on the naturals.

# Chapter 6

# Induction on Datatypes

## 6.1 Introduction

A central element in any programming algebra is the definition of datatypes as solutions of (recursive) equations. Such definitions usually include standard recursion and induction schemes over the defined datatypes.

The definition of the natural numbers is a typical example. This type can be defined inductively by stating, first, that $0$ is a natural number and, second that if $n$ is a natural number then also $\mathsf{succ}.n$ (i.e. $n + 1$) is a natural number. There are some additional requirements needed to complete the definition of the natural numbers, and one of these is that the well-known induction principle is valid.

Such an induction scheme, that comes with a newly defined datatype, is called structural induction, and is a useful tool to prove properties of the datatype. However, sometimes one encounters properties that are more readily proved using an inductive argument that is not structural. This is why alternative induction schemes have been developed for the natural numbers, for instance the induction principle known as *strong* induction. In that scheme one proves that a property holds for any natural number by assuming that it holds for all numbers smaller than an arbitrary number $n$. Under this assumption it is then proved that the property holds for $n$ as well.

That structural induction is not in all cases suitable is also indicated by the fact that one often encounters the following type of proof to establish properties of lists. Such proofs start with "We use induction on the length of the list; assume the property holds for any list shorter than $n$". Apparently, the standard structural recursion scheme was considered to be insufficient for that particular proof: what was needed is a kind of strong induction principle for lists.

Recall from chapter 3 that the strong induction principle for natural numbers is based on the fact that the relation "less than" (on natural numbers) admits induction. This suggests that a "strong" induction principle on a datatype can be obtained by generalising "admitting induction". And that is, indeed, done in this chapter: we generalise the notion "admits induction" in such a way that it can be used to prove properties of lists, while exploiting the structure of the type of lists. This way, it becomes unnecessary to use induction over the naturals in proofs of properties of lists.

The theory developed in this chapter is, in fact, general and not specific to any particular datatype. We define a notion of F-reductivity, where F stands for a "relator" (recall from chapter 2 that relators can be used to model datatypes). The notion is so called in order to avoid confusion with the existing notion of structural inductivity. We show that F-reductive relations always exist, whatever the value of F, and give laws for constructing reductive relations from existing reductive relations.

Some of the connections between reductivity, inductivity and well-foundedness are discussed. One of the main results of the chapter is that reductivity generalises the categorical notion of initiality.

The general context of this work is a relational algebra of datatypes developed in close cooperation with our own group at Eindhoven University [2], Lambert Meertens at Utrecht University and Richard Bird and Oege de Moor at Oxford University [20]. The inspiration for the notion of reductivity came from Oege de Moor during his extended visit to Eindhoven in 1992.

## 6.2 Structural induction

The first kind of induction principle for arbitrary datatypes that we discuss is the well-known structural induction. It is the standard induction scheme that is part of the definition of recursive datatypes. For instance, structural induction over the type of natural numbers is what is usually called *the* principle of induction, and its validity is one of the defining properties of the naturals.

This principle of induction can be expressed informally as: a property is true of all natural numbers if it is an *invariant* of $\mathsf{zero}\triangledown\mathsf{succ}$. By this we mean that the property is established by $\mathsf{zero}$ —a property is an "invariant" of a constant function if the result of the function satisfies the property— and the property is an invariant of $\mathsf{succ}$ if the function $\mathsf{succ}$ maps numbers satisfying the property to numbers also satisfying the property.

The question we have to tackle is how to formalise the notion of "invariance". We propose calling a monotype $A$ an *invariant* of $R$ iff

$$(R\circ F.A)< \;\subseteq A \quad .$$

Equivalently, in the predicate calculus, $A$ is an *invariant* of F-algebra $R$ iff

$$\forall(x{:}\,\exists(y{:}\; x\langle R\rangle y{:}\; y\in F.A){:}\; x\in A) \quad .$$

We call this property an invariance property because it expresses the idea that an F-structure ($y$) all of whose elements satisfy property $A$ ($y\in F.A$) is mapped by $R$ into a value ($x$) also satisfying $A$ ($x\in A$).

Our notion of a relation $R$, being "inductive" with respect to F, is that it is possible to deduce that all elements of the left domain of $R$ satisfy some property $A$ whenever $A$ is an invariant of $R$.

**Definition 6.1 (F-inductivity)** A relation $R$ is said to be F-*inductive* if, for all monotypes $A$,

$$R< \subseteq A \ \Leftarrow\ (R \circ F.A)< \subseteq A \quad . \tag{6.2}$$

□

To make clear that definition 6.1 indeed captures structural induction, we consider the datatype of lists. Let $F$ be the relator that maps $X$ to $I+(\mathbb{1}+(X\times X))$ and $R$ the function $\tau\triangledown(\mathsf{nill}\triangledown\mathsf{join})$. This last function consists of the junc of the relations $\tau$ (the function that maps a point $x$ to the singleton list $[x]$), $\mathsf{nill}$ (the constant function mapping the element of the unit type to the empty list) and $\mathsf{join}$ (the function that concatenates two lists). We now show that $R$ is F-inductive.

With the above substitutions for $F$ and $R$, the antecedent of (6.2) can be rewritten:

$$
\begin{array}{ll}
 & (\tau\triangledown(\mathsf{nill}\triangledown\mathsf{join}) \circ I+(\mathbb{1}+(A\times A)))< \subseteq A \\
\equiv & \qquad \{ \ \triangledown\text{-}+\text{-fusion} \ \} \\
 & ((\tau \circ I) \triangledown ((\mathsf{nill} \circ \mathbb{1}) \triangledown (\mathsf{join} \circ A\times A)))< \ \subseteq \ A \\
\equiv & \qquad \{ \ (X\triangledown Y)< = X< \cup Y< \ \} \\
 & (\tau \circ I)< \cup (\mathsf{nill} \circ \mathbb{1})< \cup (\mathsf{join} \circ A\times A)< \subseteq A \\
\equiv & \qquad \{ \ \mathbb{1} \text{ is a domain of } \mathsf{nill} \ \} \\
 & \tau< \subseteq A \ \wedge \ \mathsf{nill}< \subseteq A \ \wedge \ (\mathsf{join} \circ A\times A)< \subseteq A
\end{array}
$$

The first two conjuncts can be interpreted as: all singleton lists and the empty list satisfy property $A$. This is the base case of a familiar type of inductive argument. The second conjunct is the induction step: if two lists satisfy property $A$ the concatenation of the two lists also satisfies $A$. Clearly, if the base case and induction step are true, all lists satisfy $A$. This last fact can be restated as the consequent of (6.2) because the left domain of $R$ equals the type of lists: any list can be constructed as either a singleton, the empty list or the concatenation of two lists. Therefore, with this choice of $F$ and $R$ formula (6.2) holds true: the relation $\tau\triangledown(\mathsf{nill}\triangledown\mathsf{join})$ is $(X \mapsto I+(\mathbb{1}+(X\times X)))$-inductive.

If we substitute the relator $(\mathbb{1}+)$ for $F$ and the relation $\mathsf{zero}\triangledown\mathsf{succ}$ for $R$ in the antecedent of (6.2), a calculation similar to the previous one can be carried out. It can then be verified that the resulting formula is equivalent to the familiar formulation of the base case and "step" of the induction principle for the natural numbers. So, the statement that $\mathsf{zero}\triangledown\mathsf{succ}$ is $(\mathbb{1}+)$-inductive is equivalent to the statement that standard induction over the natural numbers is allowed.

## 6.3 F-reductivity

Although structural induction is important, it is certainly not the only valid way to prove facts by an inductive argument. For instance, consider the so-called strong induction principle for the natural numbers. This induction principle is just the exploitation of the fact that the standard "less than" order on the naturals admits induction. That last fact is of course equivalent to the statement that structural induction over the naturals is allowed. However, one should not conclude from this equivalence that strong induction is the same as structural induction: it is in fact a quite different proof technique.

To be able to use this different technique for induction arguments on arbitrary datatypes this section introduces a compact point-free definition of the notion of $\mathsf{F}$-*reductive* relation (where $\mathsf{F}$ is a relator). This definition generalises the notion of admitting induction by adding some extra structure in the form of a datatype.

As an example of the generalised induction principle, we consider the datatype list. Let $\mathsf{R}$ be a relation such that $m\langle \mathsf{R}\rangle x$ implies that $x$ is a natural number and $m$ is a list of natural numbers, all smaller than $x$. Now the fact that all natural numbers satisfy a certain property, call it $\mathsf{A}$, can be proved by the following inductive argument. We assume the "induction hypothesis" that for all lists $m$ which $\mathsf{R}$ relates to $x$, i.e. $m\langle \mathsf{R}\rangle x$, all of the elements of $m$ satisfy property $\mathsf{A}$. Under this assumption we prove that $x$ satisfies $\mathsf{A}$ (this proof being the induction step). That it then follows by induction that all natural numbers enjoy property $\mathsf{A}$ may or may not be easy to see, but that does not concern us at the moment. In this chapter, this example will be made more precise and also the validity of the induction principle for relation $\mathsf{R}$ is established.

What we want to show here is that such an induction principle has some relevance for program construction. Consider the following divide and conquer like procedure. The input is a natural number and the first step is to transform this number into a list of natural numbers, each smaller than the input. Then, in the second phase of the algorithm, the procedure is recursively applied to all of the numbers in the list. Finally the output of the procedure is computed from the list of results of the recursive call. That this recursive procedure terminates on all inputs (something that is also called "the recursion is well-founded") can be proved with the induction principle sketched in the previous paragraph. This claim will be made precise and proved in the next chapter.

The above discussion indicates that the fact that a relation $\mathsf{R}$ can be used in such an inductive argument is an interesting property. It therefore deserves to have a name: we call such a relation $\mathsf{F}$-*reductive*. The explanation of the name $\mathsf{F}$-reductive is the following. If $\mathsf{R}$ is an $\mathsf{F}$-reductive relation and $m\langle \mathsf{R}\rangle x$ then "output" $m$ is an $\mathsf{F}$-structure, such that all of the elements of $m$ are in a certain sense "reductions" of "input" $x$.

We now go on to formulate the notion "$\mathsf{F}$-reductivity" in a point-free way and to be able to do so we first give a formulation using points. We say that $\mathsf{R}$ is $\mathsf{F}$-reductive iff for all sets $\mathsf{A}$:

$$\forall(y:\ \forall(x:\ x\langle \mathsf{R}\rangle y\ :\ x\in \mathsf{F.A}) :\ y\in \mathsf{A})\ \Rightarrow\ \forall(y::\ y\in \mathsf{A}) \quad . \tag{6.3}$$

In words: we may conclude that all elements of the universe satisfy property $\mathsf{A}$ if we can show that any element $y$ satisfies $\mathsf{A}$ assuming the induction hypothesis that any "$\mathsf{R}$-predecessor" $x$ of $y$ is an $\mathsf{F}$-structure containing only elements satisfying $\mathsf{A}$ (i.e. $x\in \mathsf{F.A}$).

Now observe that the expression $\forall(x:\ x\langle \mathsf{R}\rangle y\ :\ x\in \mathsf{F.A})$ can be rewritten, using the characterisation of the monotype factor, as $y \in \mathsf{R}\searrow\mathsf{F.A}$. Hence, the antecedent of (6.3) reduces to $R\backslash F.A \subseteq A$. The consequent is (the understood domain of $y$ being $I$): $I \subseteq A$. Therefore the point-free formulation of (6.3) reads:

$$\forall(A:\ A \subseteq I:\ R\backslash F.A \subseteq A\ \Rightarrow\ I \subseteq A) \tag{6.4}$$

When expression (6.4) is compared with definition (3.16) of "admits induction" we see that the former indeed generalises the latter: the fact that a relation admits induction is equivalent to the fact that the relation is reductive with respect to the identity relator.

Following this definition of F-reductivity we now present a theorem giving alternative characterisations of the notion.

**Theorem 6.5 (Characterisations of F-reductivity)** The following are all equivalent.

**(a)** $\mu(R\backslash \circ F) = I$

**(b)** $\forall(A:\ A \subseteq I:\ R\backslash F.A \subseteq A\ \Rightarrow\ I \subseteq A)$

**(c)** $\forall(A:\ A \subseteq I:\ R\backslash F.A = A\ \Rightarrow\ R{>} \subseteq A)$

□

The proof is by cyclic implication. That (a) implies (b) is an immediate consequence of the Knaster-Tarski fixed point theorem. The implication (b)⇒(c) is also easy: by reflexivity of $\subseteq$ and the fact that $R{>} \subseteq I$. The more difficult step is that from (c) to (a); we show the (formally) stronger: clause (c) implies that *every* fixed point of the function $R\backslash \circ F$ is $I$. Assume :

$$R\backslash F.A = A \quad ,$$

i.e. $A$ is a fixed point of $R\backslash \circ F$, and (c). Then

$$\begin{array}{ll}
 & I = A \\
\equiv & \quad \{\ A \subseteq I,\ \text{assumption}\ \} \\
 & I \subseteq R\backslash F.A \\
\equiv & \quad \{\ \text{factors: (2.37)}\ \} \\
 & R{<} \subseteq F.A \\
\equiv & \quad \{\ \text{domains}\ \} \\
 & (R \circ R{>}){<} \subseteq F.A \\
\equiv & \quad \{\ \text{factors: (2.37)}\ \} \\
 & R{>} \subseteq R\backslash F.A \\
\equiv & \quad \{\ \text{assumption}\ \} \\
 & R{>} \subseteq A
\end{array}$$

$$\begin{array}{ll} \Leftarrow & \{ \ (c) \ \} \\ & R\searrow F.A = A \\ \equiv & \{ \ \text{assumption} \ \} \\ & \text{true} \end{array}$$

This completes the proof.

Of the three different forms of the definition of F-reductivity the first is clearly the most compact, and hence the most suited to abstract reasoning about the notion. The second form 6.5(**b**) is closest to the way proof by induction is normally presented. The third alternative, because it is formally weaker than the other two, is often useful to *prove* that a given relation is F-reductive.

To be complete, we now give the definition of reductivity. Of the formulations of theorem 6.5 we take the most compact, i.e. the first.

**Definition 6.6 (F-reductivity)** Relation R is said to be F-*reductive* if and only if it enjoys the property:

$$\mu((R\searrow)\circ F) = I \quad . \tag{6.7}$$

□

## 6.4 Towards a calculus of reductive relations

In the previous section we provided some evidence that the notion of F-reductivity is an interesting one, particularly in connection with the termination proofs of recursively defined programs. In the next three sections we give a number of results that allow us to prove that a relation is reductive. These theorems are meant to form the basis of a calculus of reductive relations. Such a calculus is of interest because it can be used in the termination proofs of programs. However, the major design criterion for the calculus is that it must be useful for the *construction* of terminating programs.

The way the calculus is intended to be used is inspired by the observation that the termination argument has a central position in the standard way of constructing repetitions. Typically, the overall shape of the body is determined by the requirement that the repetition terminates. In the same way a termination argument should suggest the form of the body of a recursive procedure.

In the next chapter it will be made clear that the termination argument of a recursive procedure boils down to showing that a certain part of the body is a reductive relation. The calculus should be useful to calculate the typical shape of such relations, i.e. it should be useful to calculate the structure of the body.

### 6.4.1 Basic F-reductive relations

In this section it is shown that for any relator $\mathsf{F}$ there exist F-reductive relations. The existence of one such relation is easy to see: we have that, for all monotypes $\mathsf{A}$, the monotype $\perp\!\!\perp\searrow\mathsf{A}$ equals the identity relation. From theorem 6.5(**c**) it then follows that the empty relation is reductive for all relators $\mathsf{F}$. However, this is a rather trivial solution and we can do better. The next theorem shows that it is always possible to turn a given relation into an F-reductive relation by restricting it by a certain monotype. The theorem is a generalisation of theorem 3.25.

**Theorem 6.8** For any relation $\mathsf{R}$ the relation $\mathsf{R}\circ\mu((\mathsf{R}\searrow)\circ\mathsf{F})$ is F-reductive.

**Proof** Let $\mathsf{B}$ be $\mu((\mathsf{R}\searrow)\circ\mathsf{F})$. By theorem 6.5 we have to show for any monotype $\mathsf{A}$:

$$(\mathsf{R}\circ\mathsf{B})\searrow\mathsf{F.A} = \mathsf{A} \;\Rightarrow\; (\mathsf{R}\circ\mathsf{B})> \subseteq \mathsf{A} \quad .$$

This is done in the following calculation.

$$\begin{array}{ll}
 & (\mathsf{R}\circ\mathsf{B})> \subseteq \mathsf{A} \\
\Leftarrow & \quad \{ \text{ domains: } (\mathsf{R}\circ\mathsf{A})> \subseteq \mathsf{A} \quad \} \\
 & \mathsf{B} \subseteq \mathsf{A} \\
\Leftarrow & \quad \{ \text{ Knaster-Tarski, definition of } \mathsf{B} \quad \} \\
 & \mathsf{R}\searrow\mathsf{F.A} \subseteq \mathsf{A} \\
\equiv & \quad \{ \text{ assume: } (\mathsf{R}\circ\mathsf{B})\searrow\mathsf{F.A} = \mathsf{A} \quad \} \\
 & \mathsf{R}\searrow\mathsf{F.A} \subseteq (\mathsf{R}\circ\mathsf{B})\searrow\mathsf{F.A} \\
\equiv & \quad \{ \text{ antimonotonicity of } \searrow\mathsf{C} \quad \} \\
 & \mathsf{R}\circ\mathsf{B} \subseteq \mathsf{R} \\
\equiv & \quad \{ \ \mathsf{B} \subseteq \mathrm{I} \quad \} \\
 & \text{true}
\end{array}$$

□

An easy consequence is that for any relator $\mathsf{F}$ there is a non-trivial F-reductive relation:

**Corollary 6.9** $\mu\mathsf{F}$ is F-reductive.

**Proof** Instantiate $\mathsf{R}$ in theorem 6.8 to $\mathrm{I}$ and observe that $(\mathrm{I}\searrow)$ is the identity.
□

### 6.4.2 New F-reductive relations from old

This section is intended to show how, given an F-reductive relation, other reductive relations can be constructed. The first way to do this is simple enough:

**Theorem 6.10** If $R$ is F-reductive and $S \subseteq R$ then $S$ is F-reductive.

**Proof** Immediate from theorem 6.5 (b) and the monotonicity properties of the monotype factor
□

The theorem is not deep, nevertheless it is extremely useful. Recall that the refinement order of programs is the same as inclusion of relations. The content of theorem 6.10 is therefore that reductivity is preserved under refinement.

The next two theorems can be used to change the "kind of reductivity", i.e. to construct F-reductive relations from G-reductive relations. These theorems formalise the idea that composing a reductive relation with a relation which "has no real effect" on a data structure results in a reductive relation. Of course, the phrase "has no real effect" is a bit vague, but the intended meaning is that such a relation transforms F-structures into G-structures without affecting the contents of the structures. That is, the only thing that can happen is that elements are copied or discarded. This was formalised in the definition of natural transformation 2.57 that is repeated here for ease of reference.

**Definition 6.11** Relation $R$ is a natural transformation from relator $F$ to relator $G$, written $R \in F \stackrel{.}{\leadsto} G$ iff for all monotypes $A$:

$$F.A \subseteq R \mathbin{\backslash\!\!\cdot}\, G.A \tag{6.12}$$

□

Using natural transformations we now can formulate the next theorem

**Theorem 6.13** Let $Q$ be G-reductive and $S \in \mathsf{Id} \stackrel{.}{\leadsto} F$, where $\mathsf{Id}$ denotes the identity relator. Then $F.Q \circ S$ is $(F{\circ}G)$-reductive

**Proof**

$$
\begin{array}{ll}
 & I \subseteq A \\
\Leftarrow & \quad \{ \quad Q \text{ is G-reductive} \quad \} \\
 & Q \mathbin{\backslash\!\!\cdot}\, G.A \subseteq A \\
\Leftarrow & \quad \{ \quad S \in I \stackrel{.}{\leadsto} F\text{, lemma 2.58} \quad \} \\
 & S \mathbin{\backslash\!\!\cdot}\, F.(Q \mathbin{\backslash\!\!\cdot}\, G.A) \subseteq A \\
\Leftarrow & \quad \{ \quad \text{factors and relators: (2.60)} \quad \} \\
 & S \mathbin{\backslash\!\!\cdot}\, (F.Q \mathbin{\backslash\!\!\cdot}\, F.G.A) \subseteq A \\
\equiv & \quad \{ \quad \text{factors: (2.47)} \quad \} \\
 & (F.Q \circ S) \mathbin{\backslash\!\!\cdot}\, F.G.A \subseteq A
\end{array}
$$

□

A typical use of theorems (6.10) and (6.13) is: $\mathsf{R}$ is $\mathsf{F}$-reductive follows from the fact that there is a well-founded relation $\mathsf{Q}$ and a relation $\mathsf{S} \in \mathsf{Id} \overset{\cdot}{\leadsto} \mathsf{F}$ such that $\mathsf{R} \subseteq \mathsf{F.Q} \circ \mathsf{S}$.

It also follows from theorem 6.13 that the relation mentioned in the introduction of the previous section is List-reductive. Recall that this relation $\mathsf{R}$ has the property that $m\langle\mathsf{R}\rangle x$ implies that $x$ is a natural number and $m$ is a list of natural numbers, all smaller than $x$. Now consider the relation $\mathsf{S}$ which relates an element $x$ of the universe to a list of arbitrary length containing only copies of $x$. This relation $\mathsf{S}$ certainly has the property $\mathsf{S} \circ \mathsf{A} \subseteq \mathsf{list.A} \circ \mathsf{S}$: if $\mathsf{S}$ is applied to an argument enjoying property $\mathsf{A}$, the result is a list and all of the elements in that list have property $\mathsf{A}$. If $\mathsf{S}$ is now composed with the relation $\mathsf{list}.<$, where $<$ is the (well-founded) less-than relation on the natural numbers, it follows that the resulting relation $\mathsf{list}.< \circ\, \mathsf{S}$ has precisely the properties of relation $\mathsf{R}$. By instantiating $\mathsf{Q}$ to $<$ and $\mathsf{G}$ to the identity relator in theorem 6.13, it follows that $\mathsf{R}$ is $\mathsf{list}$-reductive.

A particularly important $\mathsf{F}$-reductivity theorem is:

**Theorem 6.14** Let $\mathsf{R}$ be $\mathsf{F}$-reductive, $\mathsf{S} \in \mathsf{G}{\circ}\mathsf{F} \overset{\cdot}{\leadsto} \mathsf{H}{\circ}\mathsf{G}$, and $\mathsf{G}$ be a relator that distributes over all unions of monotypes. Then $\mathsf{S} \circ \mathsf{G.R}$ is $\mathsf{H}$-reductive.

**Proof** We aim to prove (c) in theorem 6.5 with the instantiations $\mathsf{R} := \mathsf{S} \circ \mathsf{G.R}$ and $\mathsf{F} := \mathsf{H}$.

$$
\begin{array}{cl}
 & (\mathsf{S} \circ \mathsf{G.R})_{>} \subseteq \mathsf{A} \\
\Leftarrow & \qquad \{ \text{ domains } \} \\
 & \mathsf{G}.I \subseteq \mathsf{A} \\
\equiv & \qquad \{ \text{ shunting. } \mathsf{G}^{\sharp} \text{ exists because } \mathsf{G} \text{ distributes over all unions } \} \\
 & I \subseteq \mathsf{G}^{\sharp}.\mathsf{A} \\
\Leftarrow & \qquad \{ \ \mathsf{R} \text{ is } \mathsf{F}\text{-reductive } \} \\
 & \mathsf{R} \searrow \mathsf{F}.(\mathsf{G}^{\sharp}.\mathsf{A}) \subseteq \mathsf{G}^{\sharp}.\mathsf{A} \\
\equiv & \qquad \{ \text{ shunting } \} \\
 & \mathsf{G}.(\mathsf{R} \searrow \mathsf{F}.(\mathsf{G}^{\sharp}.\mathsf{A})) \subseteq \mathsf{A} \\
\Leftarrow & \qquad \{ \text{ relators and factors: (2.60) } \} \\
 & \mathsf{G.R} \searrow \mathsf{G}.(\mathsf{F}.(\mathsf{G}^{\sharp}.\mathsf{A})) \subseteq \mathsf{A} \\
\Leftarrow & \qquad \{ \ \mathsf{S} \in \mathsf{G}{\circ}\mathsf{F} \overset{\cdot}{\leadsto} \mathsf{H}{\circ}\mathsf{G} \text{ ; lemma 2.58 } \} \\
 & \mathsf{G.R} \searrow (\mathsf{S} \searrow \mathsf{H}.(\mathsf{G}.(\mathsf{G}^{\sharp}.\mathsf{A}))) \subseteq \mathsf{A} \\
\Leftarrow & \qquad \{ \text{ factors: (2.47); cancellation (2.2) } \} \\
 & (\mathsf{S} \circ \mathsf{G.R}) \searrow \mathsf{H.A} \subseteq \mathsf{A}
\end{array}
$$

□

The restriction imposed on relator $\mathsf{G}$ in this theorem is rather severe. However, there is one important class of relators satisfying it: the sections $\mathsf{X} \mapsto \mathrm{I} \times \mathsf{X}$ and $\mathsf{X} \mapsto \mathsf{X} \times \mathrm{I}$ of the product relator. As will be explained in the next chapter, this enables us to give the termination proofs of programs that are defined on the product of an "inductive" domain and another domain. In other words, the theorem allows one to prove termination of programs with additional parameters. An example of such a program is the one that inserts an element in a tree. The recursion of that program is according to the structure of its second argument, the tree. The first argument, i.e. the element to be inserted, serves as a parameter that is only used in the "base case" of the recursion.

From theorem 6.14 the next result follows as a simple corollary.

**Corollary 6.15** If $\mathsf{R}$ is $\mathsf{F}$-reductive and $\mathsf{S} \in \mathsf{F} \overset{\cdot}{\rightsquigarrow} \mathsf{H}$ then $\mathsf{S} \circ \mathsf{R}$ is $\mathsf{H}$-reductive.

**Proof** Instantiate theorem 6.14 with the identity relator (which distributes, of course, over any union).
□

### 6.4.3 Bound functions

As we discussed in chapter 3 a useful technique for proving that a relation is well-founded, makes use of a bound function. We now investigate whether a similar technique can be used to prove that a relation is reductive. That is, we want to generalise theorem 3.30. We might guess that the condition

$$\mathsf{S} \subseteq \mathsf{f} \cup \circ \mathsf{R} \circ \mathsf{f} \tag{6.16}$$

together with reductivity of $\mathsf{R}$ is sufficient to guarantee reductivity of $\mathsf{S}$. However that does not seem to work out. The reason is that the righthand side of (6.16) has a typing error: it does not take into account that the left domain of an $\mathsf{F}$-reductive relation is contained in $\mathsf{F}.\mathrm{I}$.

**Lemma 6.17** For all $\mathsf{F}$-reductive relations $\mathsf{R}$: $\mathsf{R} = \mathsf{F}.\mathrm{I} \circ \mathsf{R}$ .

**Proof** The proof uses that $\mathsf{F}$-reductivity of $\mathsf{R}$ is the same as $\mu((\mathsf{R}\backslash\!\!\lrcorner\,) \circ \mathsf{F}) = \mathrm{I}$, which implies $\mathrm{I} = \mathsf{R} \backslash\!\!\lrcorner\, \mathsf{F}.\mathrm{I}$ .

$$
\begin{array}{ll}
 & \mathsf{R} \\
= & \quad \{ \ \ \mathsf{R} \text{ is } \mathsf{F}\text{-reductive, hence: } \mathrm{I} = \mathsf{R} \backslash\!\!\lrcorner\, \mathsf{F}.\mathrm{I} \ \ \} \\
 & \mathsf{R} \circ \mathsf{R} \backslash\!\!\lrcorner\, \mathsf{F}.\mathrm{I} \\
= & \quad \{ \ \ \text{monotype factors (2.42);} \\
 & \quad \ \ \ \ \mathrm{I} = \mathsf{R} \backslash\!\!\lrcorner\, \mathsf{F}.\mathrm{I} \ \ \} \\
 & \mathsf{F}.\mathrm{I} \circ \mathsf{R} \circ \mathrm{I} \\
= & \quad \{ \ \ \text{relation calculus} \ \ \} \\
 & \mathsf{F}.\mathrm{I} \circ \mathsf{R}
\end{array}
$$

□

So, according to this lemma, the possible outputs of an F-reductive relation are F-structures. But this need not be true for the possible inputs of the relation. Therefore, the function $\mathsf{f}$ in (6.16) should have a left domain which contains F-structures as well as things that are not F-structures. This explains the typing error. Therefore condition (6.16) is patched as follows:

$$\mathsf{S} \subseteq \mathsf{F.f}^{\cup} \circ \mathsf{R} \circ \mathsf{f} \tag{6.18}$$

and this last condition works well.

**Theorem 6.19** Let $\mathsf{R}$ be an F-reductive relation and $\mathsf{f}$ a functional relation. Then $\mathsf{F.f}^{\cup} \circ \mathsf{R} \circ \mathsf{f}$ is F-reductive.

**Proof** We prove that $\mathsf{F.f}^{\cup} \circ \mathsf{R} \circ \mathsf{f}$ is reductive using characterisation 6.5(**c**).

$$
\begin{array}{ll}
 & (\mathsf{F.f}^{\cup} \circ \mathsf{R} \circ \mathsf{f})_{>} \subseteq \mathsf{A} \\
\Leftarrow & \qquad \{ \ \text{domains: } (\mathsf{X} \circ \mathsf{Y})_{>} \subseteq \mathsf{Y}_{>} \ \} \\
 & \mathsf{f}_{>} \subseteq \mathsf{A} \\
\equiv & \qquad \{ \ \mathsf{f}_{>} = (\mathsf{f}^{\cup} \circ \mathrm{I})_{<}\text{; factor shunting } \} \\
 & \mathrm{I} \subseteq \mathsf{f}^{\cup} \backslash \mathsf{A} \\
\Leftarrow & \qquad \{ \ \text{assumption: } \mathsf{R} \text{ is F-reductive } \} \\
 & \mathsf{R} \backslash \mathsf{F.}(\mathsf{f}^{\cup} \backslash \mathsf{A}) \subseteq \mathsf{f}^{\cup} \backslash \mathsf{A} \\
\Leftarrow & \qquad \{ \ (2.60)\text{: } \mathsf{F.}(\mathsf{X} \backslash \mathsf{B}) \subseteq \mathsf{F.X} \backslash \mathsf{F.B} \ \} \\
 & \mathsf{R} \backslash (\mathsf{F.f}^{\cup} \backslash \mathsf{F.A}) \subseteq \mathsf{f}^{\cup} \backslash \mathsf{A} \\
\Leftarrow & \qquad \{ \ \text{assumption } \mathsf{f} \text{ is functional: } \mathsf{f} \circ \mathsf{f}^{\cup} \subseteq \mathrm{I}\ , \\
 & \qquad \quad \ \text{hence } \mathsf{B} \subseteq \mathsf{f}^{\cup} \backslash (\mathsf{f} \backslash \mathsf{B}) \ \} \\
 & \mathsf{f}^{\cup} \backslash (\mathsf{f} \backslash (\mathsf{R} \backslash (\mathsf{F.f}^{\cup} \backslash \mathsf{F.A}))) \subseteq \mathsf{f}^{\cup} \backslash \mathsf{A} \\
\Leftarrow & \qquad \{ \ \text{monotonicity of } \mathsf{f}^{\cup} \backslash \ \} \\
 & \mathsf{f} \backslash (\mathsf{R} \backslash (\mathsf{F.f}^{\cup} \backslash \mathsf{F.A})) \subseteq \mathsf{A} \\
\equiv & \qquad \{ \ \text{factors } \} \\
 & (\mathsf{F.f}^{\cup} \circ \mathsf{R} \circ \mathsf{f}) \backslash \mathsf{F.A} \subseteq \mathsf{A}
\end{array}
$$

□

It now follows by theorem 6.10 that, if $\mathsf{R}$ and $\mathsf{f}$ satisfy the conditions of theorem 6.19, condition (6.18) implies reductivity of $\mathsf{S}$.

## 6.5 Reductivity and coalgebras

In this section we show that reductivity is preserved under isomorphisms of coalgebras. Recall from chapter 2 that a relation $R$ is an algebra if $R = R \circ F.I$. The notion dual to algebra is called *coalgebra* and $R$ is called a coalgebra if the converse of $R$ is an algebra or, equivalently, $R = F.I \circ R$. So, lemma 6.17 states that any $F$-reductive relation is a coalgebra. From category theory it is known that many interesting properties of (co-)algebras are preserved under isomorphisms. (According to category theorists this preservation is necessary and sufficient for a property to be of interest.) To be able to prove that reductivity is preserved under isomorphisms we need to formulate that notion first. Two coalgebras are called isomorphic if there exists a bijection between carriers of the coalgebras which is also a homomorphism.

In chapter 2 we defined what a carrier of an algebra $R$ is: it is a monotype $A$ such that $A \circ R = R = R \circ F.A$. A carrier $A$ of a coalgebra $R$ has then of course the dual property $R \circ A = R = F.A \circ R$.

An isomorphism $f$ of coalgebras $R$ and $S$ has to be a bijection between carriers. This is the same as saying that the domains of $f$ are carriers. Therefore, we require that $f>$ is a carrier of $R$ and $f<$ is a carrier of $S$. (Of course we could have swapped the domains, but we have to choose a direction for the isomorphism.)

Now we are left with the task of formalising what a homomorphism is. Relation $f$ is a homomorphism to coalgebra $S$ from coalgebra $R$ iff:

$$S \circ f = F.f \circ R$$

By looking at some concrete examples it can be verified that this indeed captures what is usually called a homomorphism, i.e a function which distributes over the operators of a (co-)algebra. However, this is at the moment not our concern. We just want to prove the following theorem.

**Theorem 6.20** Let $f$ be an isomorphism to $F$-coalgebra $S$ from $F$-reductive relation $R$, i.e:

$$f \circ f\cup = f< \quad \wedge \quad f\cup \circ f = f> \qquad \text{(f is a bijection)}$$

$$S \circ f = F.f \circ R \qquad \text{(f is a homomorphism)}$$

$$S \circ f< = S \quad \wedge \quad R \circ f> = R \qquad \text{(f is between carriers)}$$

Then $S$ is $F$-reductive. In other words: reductivity is preserved under isomorphism of coalgebras.

**Proof** Because $f$ is a bijection, relation $f\cup$ is functional. Hence, by theorem 6.19, relation $F.f \circ R \circ f\cup$ is $F$-reductive. Thus, by theorem 6.10, $S$ is $F$-reductive follows from $S \subseteq F.f \circ R \circ f\cup$. That this is true is established as follows:

$$S \subseteq F.f \circ R \circ f^{\cup}$$

$$\equiv \quad \{ \ f \text{ is between carriers; } f^{<} = f^{\cup >} \ \}$$

$$S \circ f^{\cup >} \subseteq F.f \circ R \circ f^{\cup}$$

$$\equiv \quad \{ \ f^{\cup} \text{ is function; definition 2.32 of functions} \ \}$$

$$S \circ f^{\cup\cup} \subseteq F.f \circ R$$

$$\equiv \quad \{ \ f \text{ is a homomorphism} \ \}$$

$$\text{true}$$

□

The proof did not use all the assumptions, so in fact something stronger is proved. This suggests that a lot of generalisations can be made. Indeed, that is the case. However, this is not the place for such generalisations.

## 6.6 Connections between inductive and reductive

In this section we study some of the connections between "inductive" and "reductive". Recall that reductivity was meant to formalise strong induction, that is it should be in a sense stronger than inductivity. So we might expect that a reductive relation is also inductive. However, if we take a look at the expression $R \circ F.A$ in the definition of inductivity, it is seen that this expression is only correctly typed if the right domain of $R$ is contained in $F.I$, that is if $R$ is an algebra. So inductivity is a property of algebras. On the other hand reductivity is a property of coalgebras. So the right question to ask is: is the converse of a reductive relation inductive? This turns out to be almost true.

**Theorem 6.21** Let $R$ be an $F$-reductive relation such that $R^{<} \subseteq F.R^{>}$. Then $R^{\cup}$ is $F$-inductive.

**Proof**

$$R^{\cup <} \subseteq A$$

$$\equiv \quad \{ \ X^{<} = X^{\cup >} \ \}$$

$$R^{>} \subseteq A$$

$$\equiv \quad \{ \ \text{factors (2.37)} \ \}$$

$$I \subseteq R^{>} \backslash A$$

$$\Leftarrow \quad \{ \ R \text{ is } F\text{-reductive} \ \}$$

$$R \backslash F.(R^{>} \backslash A) \subseteq R^{>} \backslash A$$

$$\begin{array}{ll}
\Leftarrow & \{ \text{ factors (2.37) and (2.60) } \} \\
& R{>} \circ R \setminus (F.R{>} \setminus F.A) \subseteq A \\
\equiv & \{ \text{ factors (2.47) } \} \\
& R{>} \circ (F.R{>} \circ R) \setminus F.A \subseteq A \\
\equiv & \{ \text{ assumption: } R{<} \subseteq F.R{>} \} \\
& R{>} \circ R \setminus F.A \subseteq A \\
\Leftarrow & \{ \text{ factors (2.45) } \} \\
& (F.A \circ R){>} \subseteq A \\
\equiv & \{ X{<} = X{\cup}{>} \} \\
& (R{\cup} \circ F.A){<} \subseteq A
\end{array}$$

□

This theorem can also be read as: the converse of a total, reductive co-algebra is inductive. We call such a coalgebra total because $R{<} \subseteq F.R{>}$ expresses that $R{>}$ is a carrier of $R$ and therefore the coalgebra is total on the carrier. So, modulo a typing restriction, reductive relations are inductive. This restriction on the types is not a very severe one. When, in the next chapter, the role of reductive relations in program construction is studied, it will be made clear that it is even a desirable property of coalgebras.

Next, we address the question whether reductivity is really stronger than inductivity. Does there exist an inductive relation such that its converse is not reductive? To find such a counter example we first prove a theorem that gives a sufficient condition such that inductive implies reductive. The theorem can be read as: the converse of an inductive injection is reductive.

**Theorem 6.22** If $R$ is an $F$-inductive relation such that $R{\cup} \circ R \subseteq I$, then $R{\cup}$ is $F$-reductive.

**Proof**

$$\begin{array}{ll}
& R{\cup}{>} \subseteq A \\
\equiv & \{ X{<} = X{\cup}{>} \} \\
& R{<} \subseteq A \\
\Leftarrow & \{ \text{ R is F-inductive, definition 6.1 } \} \\
& (R \circ F.A){<} \subseteq A \\
\equiv & \{ X{<} = X{\cup}{>}\text{; distribution properties of } \cup \} \\
& (F.A \circ R{\cup}){>} \subseteq A \\
\equiv & \{ \text{ assumption: } R{\cup} \circ R{\cup}{\cup} \subseteq I\text{; (2.46) with } R := R{\cup} \}
\end{array}$$

$$
\begin{array}{ll}
 & \mathsf{R}^{\cup}{>} \circ \mathsf{R}^{\cup} \setminus\!\!\!\blacksquare\, \mathsf{F}.A \subseteq A \\
\Leftarrow & \{ \quad \mathsf{R}^{\cup}{>} \subseteq \mathrm{I} \quad \} \\
 & \mathsf{R}^{\cup} \setminus\!\!\!\blacksquare\, \mathsf{F}.A = A
\end{array}
$$

□

To find a relation that is inductive but not reductive we therefore have to look at non-injective inductive relations. In section 6.2 we encountered such a relation: the function $\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join})$. The converse of this relation is indeed not reductive. It is easy to check that the following holds:

$$\ll \circ \hookleftarrow^{\cup} \circ \hookleftarrow^{\cup} \in \mathsf{F} \dot{\rightsquigarrow} \mathsf{Id}$$

where $\mathsf{F}$ is the relator $\mathsf{X} \mapsto \mathrm{I} + (\mathbb{1} + (\mathsf{X} \times \mathsf{X}))$. So, if $\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join})$ were $\mathsf{F}$-inductive, relation

$$\ll \circ \hookleftarrow^{\cup} \circ \hookleftarrow^{\cup} \circ (\tau \triangledown (\mathsf{nill} \triangledown \mathsf{join}))^{\cup}$$

admits induction by corollary 6.15 and is therefore well-founded. However this relation is equal to $\ll \circ \mathsf{join}^{\cup}$, a relation that relates the empty list to the empty list, hence it is not well-founded.

To conclude this section, we give another example of an inductive relation. Because $\mu\mathsf{F}$ is a monotype and a fixed point of $\mathsf{F}$, we have $\mu\mathsf{F}{<} \subseteq \mathsf{F} \,.\, \mu\mathsf{F}{>}$. So we get from theorem 6.21 and corollary 6.9 the following result.

**Corollary 6.23** For any relator $\mathsf{F}$, the relation $\mu\mathsf{F}$ is $\mathsf{F}$-inductive.
□

## 6.7 Well-foundedness generalised

Now that "admits induction" has been generalised we are interested in what the generalisation of "well-founded" might be. Considering the way reductivity is obtained from admits induction, i.e. by adding a relator, the first guess would be: relation $\mathsf{R}$ is $\mathsf{F}$-well-founded if $\nu(\mathsf{X} \mapsto \mathsf{F}.\mathsf{X} \circ \mathsf{R}) = \perp\!\!\!\perp$. However, that does not seem to be a reasonable definition. For instance, it can not be shown that, with this definition, $\mathsf{F}$-reductivity implies $\mathsf{F}$-well-foundedness. The main problem with the definition is that a fixed point $\mathsf{Y}$ of the function $\mathsf{X} \mapsto \mathsf{F}.\mathsf{X} \circ \mathsf{R}$ has the property $\mathsf{Y}{<} = \mathsf{F}.\mathsf{Y}{<}$ : its left domain is a fixed point of $\mathsf{F}$. In systems related to ours, such as category theory or typed relation algebras, however, such fixed points do not necessarily exist. And therefore, if we take fixed points of the function $\mathsf{X} \mapsto \mathsf{F}.\mathsf{X} \circ \mathsf{R}$ we make use of a special feature of the particular system in which we choose to work. When one uses such a feature some care has to be taken to ensure that the result has still the intended link to reality. After all, systems like category theory are also meant to model the "reality". And if something can be done in one system which, although mathematically correct, does not make sense in a related system, that might very well be an indication that the connection to the real world is lost. This seems to be the case here: maybe the feature of our system is a "bug".

A definition of F-well-foundedness that does make sense, and works also for the related systems, reads as follows.

**Definition 6.24 (F-well-founded)** Relation R is F-well-founded iff for all relations S and X:

$$X = S \circ F.X \circ R \quad \equiv \quad X = \mu(Y \mapsto S \circ F.Y \circ R)$$

□

As an indication of the relevance of this definition, we remark that the lefthand side of this equivalence, seen as an equation in X, corresponds to an interesting recursion scheme. The definition is equivalent to stating that such recursive equations have a unique solution and, therefore, can be used to define a relation. This topic is discussed further in the next chapter.

A question that can be asked now is whether definition 6.24 is indeed a generalisation of well-foundedness. That is are "well-founded" and Id-well-founded (recall Id is the identity relator) equivalent notions? The answer to this question is positive. One implication is immediate. Suppose relation R is Id-well-founded. When we substitute for S the identity relation, definition 6.24 gives

$$X = X \circ R \quad \Rightarrow \quad X = \perp\!\!\perp \qquad .$$

(Because $\perp\!\!\perp$ is the zero of composition relation $\mu(Y \mapsto Y \circ R)$ is equal to $\perp\!\!\perp$.) Comparing this with definition 3.5 of well-foundedness we see that Id-well-foundedness implies well-foundedness. For the other implication a bit more work has to be done.

First note that (for the identity relator) definition 6.24 can be restated as "the least fixed point of function $Y \mapsto S \circ Y \circ R$ is the unique fixed point of that function". The least fixed point is of course $\perp\!\!\perp$, so we must show:

$$X = S \circ X \circ R \quad \Rightarrow \quad X = \perp\!\!\perp \qquad .$$

This under the assumption that R is well-founded.

$$\begin{array}{ll}
& X = S \circ X \circ R \\
\Rightarrow & \qquad \{ \text{ Leibniz } \} \\
& \top\!\!\top \circ X = \top\!\!\top \circ S \circ X \circ R \\
\Rightarrow & \qquad \{ \ \top\!\!\top \circ S \subseteq \top\!\!\top \ \} \\
& \top\!\!\top \circ X \subseteq \top\!\!\top \circ X \circ R \\
\Rightarrow & \qquad \{ \text{ assumption: } R \text{ is well-founded (3.6) } \} \\
& \top\!\!\top \circ X \subseteq \perp\!\!\perp
\end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ domains: } \top\!\!\top \circ \mathsf{X} = \top\!\!\top \circ \mathsf{X}{>}; \ \mathrm{I} \subseteq \top\!\!\top \ \} \\
& \mathsf{X}{>} \subseteq \perp\!\!\perp \\
= & \{ \text{ domains: } \mathsf{X}{>}{=}\perp\!\!\perp \ \equiv \ \mathsf{X} = \perp\!\!\perp \ \} \\
& \mathsf{X} = \perp\!\!\perp
\end{array}
$$

Indeed: $\mathsf{Id}$-well-foundedness also follows from well-foundedness, hence the two are equivalent and $\mathsf{F}$-well-foundedness is a proper generalisation of well-foundedness.

Next we show that the property that reductivity implies well-foundedness goes through for the generalised notions. In other words: if $\mathsf{R}$ is an $\mathsf{F}$-reductive relation then, for any relation $\mathsf{S}$, the function $\mathsf{Y} \mapsto \mathsf{S} \circ \mathsf{F.Y} \circ \mathsf{R}$ has a unique fixed point. This, in turn, is equivalent to: every fixed point is contained in the least fixed point. So we assume that $\mathsf{X}$ and $\mathsf{Z}$ are an arbitrary fixed point and the least fixed point of $\mathsf{Y} \mapsto \mathsf{S} \circ \mathsf{F.Y} \circ \mathsf{R}$, respectively. We furthermore assume $\mathsf{R}$ is $\mathsf{F}$-reductive.

$$
\begin{array}{ll}
& \mathsf{X} \subseteq \mathsf{Z} \\
\equiv & \{ \text{ assumption: } \mathsf{R} \text{ is } \mathsf{F}\text{-reductive, i.e } \mu(\mathsf{R}\backslash \circ \mathsf{F}) = \mathrm{I} \ \} \\
& \mathsf{X} \circ \mu(\mathsf{R}\backslash \circ \mathsf{F}) \ \subseteq \ \mathsf{Z} \\
\Leftarrow & \{ \ \mu\text{-fusion: (2.4) ;} \\
& \quad \text{assumption: } \mathsf{Z} \text{ is least fixed point;} \\
& \quad \text{for any } \mathsf{A} \ \} \\
& \mathsf{X} \circ \mathsf{R}\backslash \mathsf{F.A} \ \subseteq \ \mathsf{S} \circ \mathsf{F.}(\mathsf{X} \circ \mathsf{A}) \circ \mathsf{R} \\
\equiv & \{ \text{ assumption: } \mathsf{X} \text{ is a fixed point;} \\
& \quad \mathsf{F} \text{ distributes over composition } \} \\
& \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R} \circ \mathsf{R}\backslash \mathsf{F.A} \ \subseteq \ \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{F.A} \circ \mathsf{R} \\
\Leftarrow & \{ \text{ monotonicity } \} \\
& \mathsf{R} \circ \mathsf{R}\backslash \mathsf{F.A} \ \subseteq \ \mathsf{F.A} \circ \mathsf{R} \\
\equiv & \{ \text{ factors: (2.43) } \} \\
& \text{true}
\end{array}
$$

This completes the proof of the following theorem.

**Theorem 6.25** An $\mathsf{F}$-reductive relation is $\mathsf{F}$-well-founded.
□

An immediate corollary of this theorem and corollary 6.23 is that the least fixed point of a relator is reductive.

**Corollary 6.26** For any relator $\mathsf{F}$, relation $\mu\mathsf{F}$ is $\mathsf{F}$-well-founded.
□

For the identity relator it is the case that "admitting induction" and "well-founded" are equivalent notions. This is, however, not the case for the generalisations F-reductive and F-well-founded, as is demonstrated by the following counter example. Let $F.X = X{\times}X$ and consider the relation $R{\vartriangle}I$ where $R$ is a non-empty but (Id-)well-founded relation. First we demonstrate that $R{\vartriangle}I$ is F-well-founded by showing that any fixed point of the function $Y \mapsto S\circ F.Y\circ R{\vartriangle}I$ is equal to the least fixed point of that function.

$$
\begin{array}{ll}
 & X = S\circ F.X\circ R{\vartriangle}I \\
\Rightarrow & \quad \{ \ \text{Leibniz; definition of } F; \\
 & \quad\quad \times\text{-}{\vartriangle}\text{-fusion} \ \} \\
 & \top\!\!\top\circ X = \top\!\!\top\circ S\circ (X\circ R){\vartriangle}X \\
\Rightarrow & \quad \{ \ \top\!\!\top\circ S\subseteq \top\!\!\top \ \} \\
 & \top\!\!\top\circ X \subseteq \top\!\!\top\circ (X\circ R){\vartriangle}X \\
\equiv & \quad \{ \ \text{domains: } \top\!\!\top\circ Y = \top\!\!\top\circ Y{>} \ \} \\
 & \top\!\!\top\circ X \subseteq \top\!\!\top\circ ((X\circ R){\vartriangle}X){>} \\
\Rightarrow & \quad \{ \ \text{domains: } (Y{\vartriangle}Z){>} = Y{>}\cap Z{>} \ \} \\
 & \top\!\!\top\circ X \subseteq \top\!\!\top\circ (X\circ R){>} \\
\equiv & \quad \{ \ \text{domains: } \top\!\!\top\circ Y = \top\!\!\top\circ Y{>} \ \} \\
 & \top\!\!\top\circ X \subseteq \top\!\!\top\circ X\circ R \\
\Rightarrow & \quad \{ \ \text{assumption: } R \text{ well-founded} \ \} \\
 & \top\!\!\top\circ X = \perp\!\!\perp \\
\equiv & \quad \{ \ \text{as before} \ \} \\
 & X = \perp\!\!\perp \\
\equiv & \quad \{ \ F \text{ is } \perp\!\!\perp\text{-strict} \ \} \\
 & X = \mu(Y \mapsto S\circ F.Y\circ R{\vartriangle}I)
\end{array}
$$

On the other hand $R{\vartriangle}I$ is certainly not F-reductive, something we show by contradiction.

$$
\begin{array}{ll}
 & R{\vartriangle}I \text{ is F-reductive} \\
\Rightarrow & \quad \{ \ \gg\in F\dot{\rightsquigarrow}\mathsf{Id}\text{; Corollary 6.15;} \\
 & \quad\quad \mathsf{id}\text{-reductive implies well-founded} \ \} \\
 & \gg\circ R{\vartriangle}I \text{ is well-founded} \\
\equiv & \quad \{ \ \gg\circ R{\vartriangle}I \ = \ I\circ R{>} \ \} \\
 & R{>} \text{ is well-founded}
\end{array}
$$

$$\equiv \qquad \{ \text{ The only well-founded monotype is } \perp\!\!\perp \ \}$$
$$R> = \perp\!\!\perp$$

However, by assumption R, and therefore R>, differ from bottom. Hence R△I is not F-reductive.

## 6.8 Initial algebras

From category theory (or rather universal algebra) the notion *initial algebra* is well-known. Initial algebras arise if one defines inductive datatypes such as natural numbers or lists. Their carrier is the newly constructed type and the operations are the constructors of the type. For instance for the natural numbers the operations are the constant function 0 and the successor function. A property of initial algebras is that they are inductive: they can be used for structural induction over their carrier.

The particular interest of this notion for computing science stems from the fact that a certain class of recursively defined programs can be described in terms of initial algebras. This aspect is discussed in the next chapter. The purpose of introducing initial algebras here is to give more evidence for the relevance of F-reductivity by relating this concept to initial algebras.

The usual definition of initial algebra in category theory texts is stated in terms of diagrams and the existence of unique arrows making those diagrams commute. For our purposes, a disadvantage of that approach is that it does not lend itself to equational (or calculational) reasoning. To be able to calculate with the notion we first disregard the diagrams altogether and rephrase the categorical description as follows. For any F-algebra there exists a unique homomorphism from the initial F-algebra to that arbitrary F-algebra. The trick is then to introduce a function from algebras to homomorphisms, denoted by the "banana brackets" $(\![\ ]\!)$, such that for algebra R the relation $(\![R]\!)$ is the unique homomorphism from the initial algebra. ("Banana brackets" were introduced by Malcolm [62].)

This approach is taken for instance by Fokkinga [42] and his definition of initial algebra can almost directly be used to define the notion in the relation calculus. A minor difficulty is caused by the fact that the relation calculus is, as opposed to category theory, an untyped system. In category theory, composition is a partial operator and when a composition is written the fact that its operands are of the right type (i.e. that the composition is defined) is usually left implicit. In relation calculus on the other hand, composition is total. Therefore, type information which is left implicit in the formulae of the categorical definition, has to be made explicit in the definition in the relation calculus. The fact that it is necessary to have explicit type information in the formulae is a disadvantage of using an untyped system. However, a great advantage of working in an untyped relation algebra is that we need not worry about the existence of initial algebras. As we will see, any relator F has an initial algebra, namely the relation $\mu F$. When working in a typed system, such as category theory, relators are not endofunctions on a complete lattice. Therefore, in such systems relators do not necessarily have least fixed points and, consequently, more work has to be done to establish the existence of initial algebras.

Adding type information to Fokkinga's definition, we propose the following definition of initial algebra.

**Definition 6.27 (Initial algebra)** A relation $\varphi$ is called an *initial* F-*algebra* iff it satisfies the following properties. First,

$$\varphi \;=\; \varphi \circ \mathsf{F}.\varphi_{<} \tag{6.28}$$

Furthermore, there exists a function, denoted by $(\![\ ]\!)$, from relations to relations such that for all relations X and R:

$$\mathsf{X}\circ\varphi = \mathsf{R}\circ\mathsf{F.X} \;\wedge\; \mathsf{X} = \mathsf{X}\circ\varphi_{<} \quad\equiv\quad \mathsf{X} = (\![\mathsf{R}]\!) \tag{6.29}$$

□

The relation $(\![\mathsf{R}]\!)$ is called the *catamorphism* of R. The type information stated in (6.28) can be read as: $\varphi$ is an algebra with carrier $\varphi_{<}$. Conjunct $\mathsf{X}\circ\varphi = \mathsf{R}\circ\mathsf{F.X}$ is the statement that X is a homomorphism from $\varphi$ to R . Expression $\mathsf{X} = \mathsf{X}\circ\varphi_{<}$ in (6.29) is to be interpreted as: relation X is defined on the carrier of $\varphi$ only. If this type information is left out we obtain the definition of [42]. Note that (6.28) can be read as: there is a unique homomorphism from $\varphi$ to R.

With this definition, it is not difficult to relate initial algebras and F-well-founded bijections.

**Lemma 6.30** Let $\varphi$ be a relation such that $\varphi\circ\varphi\cup = \varphi_{<}$ and $\varphi\cup\circ\varphi = \mathsf{F}\,.\,\varphi_{<}$. Let, furthermore, $\varphi\cup$ be F-well-founded. Then $\varphi$ is an initial F-algebra.

**Proof** First note that the assumption $\varphi\cup\circ\varphi = \mathsf{F}.\varphi_{<}$ states that the right domain of $\varphi$ (i.e. $\varphi\cup\circ\varphi \cap \mathrm{I}$) equals $\mathsf{F}.\varphi_{<}$. Therefore it is certainly the case that $\varphi \;=\; \varphi \circ \mathsf{F}.\varphi_{<}$: requirement (6.28) of the definition of initial algebra is fulfilled. So we are left with the obligation to show the second requirement, i.e. the existence of a function $(\![\ ]\!)$ such that for all X and R:

$$\mathsf{X}\circ\varphi = \mathsf{R}\circ\mathsf{F.X} \;\wedge\; \mathsf{X} = \mathsf{X}\circ\varphi_{<} \quad\equiv\quad \mathsf{X} = (\![\mathsf{R}]\!) \qquad .$$

Because $\varphi\cup$ is F-well-founded the candidate for $(\![\mathsf{R}]\!)$ is the unique solution of the equation in X:

$$\mathsf{X} = \mathsf{R}\circ\mathsf{F.X}\circ\varphi\cup \qquad .$$

(The unique solution is the least fixed point of the function $(\mathsf{Y} \mapsto \mathsf{R}\circ\mathsf{F.Y}\circ\varphi\cup)$.) So, we are done if we can show that:

$$\mathsf{X}\circ\varphi = \mathsf{R}\circ\mathsf{F.X} \;\wedge\; \mathsf{X} = \mathsf{X}\circ\varphi_{<} \quad\equiv\quad \mathsf{X} = \mathsf{R}\circ\mathsf{F.X}\circ\varphi\cup \qquad . \tag{6.31}$$

This is indeed the case: it is a consequence of the fact that $\varphi$ is a bijection. The proof is by mutual implication.

$$
\begin{array}{ll}
& X \circ \varphi = R \circ F.X \;\wedge\; X = X \circ \varphi_< \\
\Rightarrow & \quad \{ \text{ Leibniz: composition is a function } \} \\
& X \circ \varphi \circ \varphi^\cup = R \circ F.X \circ \varphi^\cup \;\wedge\; X = X \circ \varphi_< \\
\Rightarrow & \quad \{ \text{ assumption: } \varphi \circ \varphi^\cup = \varphi_< \ \} \\
& X = R \circ F.X \circ \varphi^\cup \\
\Rightarrow & \quad \{ \text{ Leibniz: composition is a function;} \\
& \qquad X = R \circ F.X \circ \varphi^\cup \text{ therefore } X = X \circ \varphi_< \ \} \\
& X \circ \varphi = R \circ F.X \circ \varphi^\cup \circ \varphi \;\wedge\; X = X \circ \varphi_< \\
\equiv & \quad \{ \text{ assumption: } \varphi^\cup \circ \varphi = F\,.\,\varphi_<; \\
& \qquad F \text{ distributes over composition } \} \\
& X \circ \varphi = R \circ F.(X \circ \varphi_<) \;\wedge\; X = X \circ \varphi_< \\
\equiv & \quad \{ \text{ trivial } \} \\
& X \circ \varphi = R \circ F.X \;\wedge\; X = X \circ \varphi_<
\end{array}
$$

□

An alternative phrasing of this theorem is: the converse of an $F$-well-founded bijective coalgebra is an initial algebra. Because of theorem 6.25 (a reductive relation is well-founded) the previous theorem 6.30 has as a corollary that also the converse of a reductive bijective $F$-coalgebra is an initial algebra.

**Corollary 6.32** Let $\varphi$ be a relation such that $\varphi \circ \varphi^\cup = \varphi_<$ and $\varphi^\cup \circ \varphi = F\,.\,\varphi_<$. Let, furthermore, $\varphi^\cup$ be $F$-reductive. Then $\varphi$ is an initial $F$-algebra.
□

Another corollary is that the least fixed point of a relator $F$ is an initial algebra. Corollary 6.9 states that $\mu F$ is $F$-reductive. Now the conditions of corollary 6.32 are readily checked because $\mu F$ is a monotype.

**Corollary 6.33** For any relator $F$ the relation $\mu F$ is an initial algebra.
□

## 6.9 Lambek's lemma

From definition (6.27) a lot of other interesting properties can be derived. What we aim for in this section is to prove that an initial $F$-algebra is a bijection between its carrier $F$ applied to the carrier. Then, in the next section, it is shown that the converse of an initial algebra is reductive. From these results it is possible to prove that, for bijections, the notions reductivity, well-foundedness, inductivity and initiality are all the same.

It is a well-known result from category theory (known as Lambek's lemma [59]) that an initial algebra is indeed a bijection. We could therefore give a proof "by reference". However that would not be totally safe because our definition of initiality differs slightly from the categorical one. Recall that because we work in an untyped system we had to add type information to the definition. To make absolutely sure that Lambek's lemma also holds in our untyped system we provide a proof, instead of relying on the literature. And we want to be absolutely sure because we consider the result that initiality and reductivity are the same notion for bijections important.

In the proof of Lambek's lemma we follow [42]. The proof is long but, on the other hand it consists of nice and short calculations. Let $\varphi$ be an initial $\mathsf{F}$-algebra. First note that by instantiating $\mathsf{X}$ in the righthand side of (6.29) to $(\![\mathsf{R}]\!)$ we obtain

$$(\![\mathsf{R}]\!) = (\![\mathsf{R}]\!) \circ \varphi{<} \quad \text{and} \tag{6.34}$$

$$(\![\mathsf{R}]\!) \circ \varphi = \mathsf{R} \circ \mathsf{F}.(\![\mathsf{R}]\!) \quad . \tag{6.35}$$

The first states that the right domain of $(\![\mathsf{R}]\!)$ is contained in the left domain of $\varphi$, whereas the second one is referred to as the *computation rule*. Furthermore, we have the *fusion rule*:

$$\mathsf{R} \circ (\![\mathsf{S}]\!) = (\![\mathsf{T}]\!) \;\Leftarrow\; \mathsf{R} \circ \mathsf{S} = \mathsf{T} \circ \mathsf{F}.\mathsf{R} \quad . \tag{6.36}$$

This rule is proved as follows.

$$\begin{array}{cl}
& \mathsf{R} \circ (\![\mathsf{S}]\!) = (\![\mathsf{T}]\!) \\
\equiv & \qquad \{ \;\; (6.29) \;\; \} \\
& \mathsf{R} \circ (\![\mathsf{S}]\!) \circ \varphi = \mathsf{T} \circ \mathsf{F}.(\mathsf{R} \circ (\![\mathsf{S}]\!)) \;\;\wedge\;\; \mathsf{R} \circ (\![\mathsf{S}]\!) = \mathsf{R} \circ (\![\mathsf{S}]\!) \circ \varphi{<} \\
\equiv & \qquad \{ \;\; \text{computation rule (6.35); (6.34)} \;\; \} \\
& \mathsf{R} \circ \mathsf{S} \circ \mathsf{F}.(\![\mathsf{S}]\!) = \mathsf{T} \circ \mathsf{F}.\mathsf{R} \circ \mathsf{F}.(\![\mathsf{S}]\!) \\
\Leftarrow & \qquad \{ \;\; \text{Leibniz} \;\; \} \\
& \mathsf{R} \circ \mathsf{S} = \mathsf{T} \circ \mathsf{F}.\mathsf{R}
\end{array}$$

Using the computation and fusion rules it is now possible to show how to construct the inverse of $\varphi$ in terms of catamorphisms but without using the converse operator. As a first result we have that the left domain of $\varphi$ equals the catamorphism of $\varphi$.

$$\varphi{<} = (\![\varphi]\!) \quad . \tag{6.37}$$

The proof of this is easy:

$$\begin{array}{cl}
& \varphi{<} = (\![\varphi]\!) \\
\equiv & \qquad \{ \;\; (6.29) \;\; \} \\
& \varphi{<} \circ \varphi \;=\; \varphi \circ \mathsf{F}.\varphi{<} \;\;\wedge\;\; \varphi{<} = \varphi{<} \circ \varphi{<} \\
\equiv & \qquad \{ \;\; \text{domains and (6.28); for all monotypes } \mathsf{A} \circ \mathsf{A} = \mathsf{A} \;\; \} \\
& \text{true}
\end{array}$$

The result (6.37) is now used to give another expression for the carrier $\varphi<$.

$$\varphi \circ ([\mathsf{F}.\varphi]) = \varphi< \tag{6.38}$$

$$
\begin{array}{ll}
 & \varphi \circ ([\mathsf{F}.\varphi]) = \varphi< \\
\equiv & \quad \{ \ (6.37) \ \} \\
 & \varphi \circ ([\mathsf{F}.\varphi]) = ([\varphi]) \\
\Leftarrow & \quad \{ \ \text{fusion } (6.36) \ \} \\
 & \varphi \circ \mathsf{F}.\varphi = \varphi \circ \mathsf{F}.\varphi \\
\equiv & \quad \{ \ \text{reflexivity} \ \} \\
 & \text{true}
\end{array}
$$

We go on by showing that relator $\mathsf{F}$ applied to the carrier is equal to the composition of the catamorphism of $\mathsf{F}$ applied to the initial algebra and the initial algebra.

$$([\mathsf{F}.\varphi]) \circ \varphi = \mathsf{F}.\varphi< \tag{6.39}$$

$$
\begin{array}{ll}
 & ([\mathsf{F}.\varphi]) \circ \varphi \\
= & \quad \{ \ \text{computation rule } (6.35) \ \} \\
 & \mathsf{F}.\varphi \circ \mathsf{F}.([\mathsf{F}.\varphi]) \\
= & \quad \{ \ \text{relators distribute over composition} \ \} \\
 & \mathsf{F}.(\varphi \circ ([\mathsf{F}.\varphi])) \\
= & \quad \{ \ (6.38) \ \} \\
 & \mathsf{F}.\varphi<
\end{array}
$$

So, $\varphi<$ and $\mathsf{F}.\varphi<$ can be obtained by the two different ways of composing the relations $([\mathsf{F}.\varphi])$ and $\varphi$. We want to show that the initial algebra is a bijection: $\varphi \circ \varphi\cup = \varphi<$ and $\varphi\cup \circ \varphi = \mathsf{F}.\varphi<$. This suggest that we try to prove $\varphi\cup = ([\mathsf{F}.\varphi])$.

$$
\begin{array}{ll}
 & \varphi\cup \\
= & \quad \{ \ \text{definition: } (6.28) \ \} \\
 & \mathsf{F}.\varphi< \circ \varphi\cup \\
= & \quad \{ \ (6.39) \ \} \\
 & ([\mathsf{F}.\varphi]) \circ \varphi \circ \varphi\cup \\
\supseteq & \quad \{ \ \text{domains } \mathsf{R} \circ \mathsf{R}\cup \supseteq \mathsf{R}< \ \} \\
 & ([\mathsf{F}.\varphi]) \circ \varphi< \\
= & \quad \{ \ (6.34) \ \} \\
 & ([\mathsf{F}.\varphi])
\end{array}
$$

For the other inclusion we calculate:

$$
\begin{array}{ll}
 & (\![\mathsf{F}.\varphi]\!) \\
= & \quad \{ \quad (6.34) \quad \} \\
 & (\![\mathsf{F}.\varphi]\!) \circ \varphi{<} \\
= & \quad \{ \quad (6.38) \text{ and converse} \quad \} \\
 & (\![\mathsf{F}.\varphi]\!) \circ (\![\mathsf{F}.\varphi]\!)^{\cup} \circ \varphi^{\cup} \\
\supseteq & \quad \{ \quad \text{domains: } \mathsf{R} \circ \mathsf{R}^{\cup} \supseteq \mathsf{R}{<} \quad \} \\
 & (\![\mathsf{F}.\varphi]\!){<} \circ \varphi^{\cup} \\
= & \quad \{ \quad \text{see below: } \varphi{>} \subseteq (\![\mathsf{F}.\varphi]\!){<} \quad \} \\
 & \varphi^{\cup}
\end{array}
$$

To justify the last step:

$$
\begin{array}{ll}
 & (\![\mathsf{F}.\varphi]\!){<} \\
\supseteq & \quad \{ \quad \text{domains: } \mathsf{R}{<} \supseteq (\mathsf{R} \circ \mathsf{S}){<}\text{; this follows from (2.31)} \quad \} \\
 & ((\![\mathsf{F}.\varphi]\!) \circ \varphi){<} \\
= & \quad \{ \quad (6.39) \quad \} \\
 & \mathsf{F}.\varphi{<} \\
\supseteq & \quad \{ \quad \text{definition (6.28); domains:} \\
 & \qquad\quad \mathsf{R} \circ \mathsf{A} = \mathsf{R} \ \Rightarrow\ \mathsf{R}{>} \subseteq \mathsf{A} \quad \} \\
 & \varphi{>}
\end{array}
$$

This completes the proof of

$$(\![\mathsf{F}.\varphi]\!) = \varphi^{\cup} \quad . \tag{6.40}$$

Together with (6.38) and (6.39) this gives us Lambek's lemma as a result.

**Lemma 6.41 (Lambek)** An initial F-algebra is a bijection:

$$\varphi \circ \varphi^{\cup} = \varphi{<} \quad \wedge \quad \varphi^{\cup} \circ \varphi = \mathsf{F}\,.\,\varphi{<} \tag{6.42}$$

□

## 6.10 Initiality implies reductivity

We have already established that the converse of a bijective reductive relation is an initial algebra (see corollary 6.32). In this section the converse of that result is proved: it is also the case that the converse of an initial algebra is F-reductive. We prove the result using characterisation 6.5(c) of reductivity. Let $\varphi$ be an initial algebra and $A$ an arbitrary monotype. We must show:

$$(\varphi^{\cup})_{>} \subseteq A \quad \Leftarrow \quad A \subseteq \varphi^{\cup} \searrow F.A \qquad . \tag{6.43}$$

We start with the consequence.

$$
\begin{array}{cl}
 & (\varphi^{\cup})_{>} \subseteq A \\
\equiv & \qquad \{ \quad \text{domains } (R^{\cup})_{>} = R_{<} \quad \} \\
 & \varphi_{<} \subseteq A \\
\equiv & \qquad \{ \quad \text{for monotypes intersection is the same as composition} \quad \} \\
 & \varphi_{<} \circ A = \varphi_{<} \\
\equiv & \qquad \{ \quad (6.37) \quad \} \\
 & \varphi_{<} \circ A = (\![\varphi]\!) \\
\equiv & \qquad \{ \quad \text{definition of initial algebra (6.29)} \quad \} \\
 & \varphi_{<} \circ A \circ \varphi = \varphi \circ F.(\varphi_{<} \circ A) \;\; \wedge \;\; \varphi_{<} \circ A = \varphi_{<} \circ A \circ \varphi_{<} \\
\equiv & \qquad \{ \quad \text{composition of monotypes is symmetric; domains} \\
 & \qquad \quad\;\; \text{relators distribute over composition; (6.28)} \quad \} \\
 & A \circ \varphi = \varphi \circ F.A
\end{array}
$$

That the last line is implied by $A \subseteq \varphi^{\cup} \searrow F.A$ is proved by rewriting the last line as two inclusions. For the first inclusion:

$$
\begin{array}{cl}
 & A \circ \varphi \subseteq \varphi \circ F.A \\
\equiv & \qquad \{ \quad \text{domains} \quad \} \\
 & (A \circ \varphi)_{>} \subseteq F.A \\
\equiv & \qquad \{ \quad \text{domains} \quad \} \\
 & (\varphi^{\cup} \circ A)_{<} \subseteq F.A \\
\equiv & \qquad \{ \quad \text{monotype factor shunting} \quad \} \\
 & A \subseteq \varphi^{\cup} \searrow F.A
\end{array}
$$

Next the other inclusion.

$$
\begin{array}{cl}
 & \varphi \circ F.A \subseteq A \circ \varphi \\
\equiv & \qquad \{ \quad \text{domains} \quad \} \\
 & (\varphi \circ F.A)_{<} \subseteq A \\
\equiv & \qquad \{ \quad \text{domains} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
& (\mathsf{F.A} \circ \varphi^{\cup})> \subseteq \mathsf{A} \\
\equiv & \{ \text{ monotype factor (2.46), } \varphi^{\cup} \text{ is a function } \} \\
& \varphi^{\cup}> \circ\, \varphi^{\cup} \backslash\!\!\!\backslash\, \mathsf{F.A} \subseteq \mathsf{A} \\
\Leftarrow & \{ \text{ composition is intersection on monotypes.} \\
& \quad \text{monotonicity of intersection } \} \\
& \varphi^{\cup} \backslash\!\!\!\backslash\, \mathsf{F.A} \subseteq \mathsf{A}
\end{array}
$$

Now the validity of (6.43) has been established and the proof of the desired theorem is completed.

**Theorem 6.44** The converse of an initial F-algebra is F-reductive.
□

## 6.11 Factorisations of initiality

Now we have all the ingredients to prove the following theorem, one of the main results of this chapter. It states that initiality can be "factored" into more fundamental concepts.

The idea of factoring initiality into, i.e. writing it as the conjunction of, other concepts is of course not new. However the way this is usually done is by remarking that an initial algebra is a bijection which is also inductive. Recall from chapter 2 that the way to construct recursive datatypes (or, which is the same thing, initial algebras) is by first postulating a "no junk and no confusion" property, i.e. the bijectivity requirement. The other ingredient of the definition is the requirement that structural induction be possible. However, the theorem we present here shows that there are other ways to factor initiality.

**Theorem 6.45** All of the following are equivalent

1. $\varphi^{\cup}$ is F-reductive $\wedge$ $\varphi \circ \varphi^{\cup} = \varphi<$ $\wedge$ $\varphi^{\cup} \circ \varphi = \mathsf{F}.\varphi<$
2. $\varphi$ is F-inductive $\wedge$ $\varphi \circ \varphi^{\cup} = \varphi<$ $\wedge$ $\varphi^{\cup} \circ \varphi = \mathsf{F}.\varphi<$
3. $\varphi^{\cup}$ is F-well-founded $\wedge$ $\varphi \circ \varphi^{\cup} = \varphi<$ $\wedge$ $\varphi^{\cup} \circ \varphi = \mathsf{F}.\varphi<$
4. $\varphi$ is an initial F-algebra

**Proof** The equivalence of the first two statements follows from theorems 6.21 and 6.22. The equivalence of the first, third and fourth statement is shown by cyclic implication. The first implies the third because of theorem 6.25. The implication from the third to the fourth is lemma 6.30. Finally, the first statement follows from the last by the combination of theorem 6.44 and Lambek's lemma (6.41).
□

Because for us reductivity is more important than inductivity and well-foundedness we are particularly interested in the equivalence of 1 and 4. It is clear that this equivalence means that initiality can be "factored" into bijectivity and reductivity. And it seems that a lot of well-known properties of initial algebras are in fact properties that hold for any reductive relation. A very important such property is the fact that well-founded recursions can be constructed using initial algebras. An important property indeed because it is one of the main reasons why computing scientists got interested in initiality and category theory.

Other properties that hold for reductive relations in general rather than just for initial algebras can be found by studying the well-known methods of constructing new initial algebras from old. It is more often than not the case that such methods are in fact ways to construct new reductive relations from old, ways which happen to preserve bijectivity. One can try to generalise such methods by relaxing the requirement that bijections have to be preserved. This often leads to new methods of constructing reductive relations. The construction methods presented in section 6.4.2 are typical examples of this.

## 6.12 Reductivity and well-foundedness

Although $\mathsf{F}$-reductivity and $\mathsf{F}$-well-foundedness are not the same, one expects that there is a strong connection between reductivity and $\mathsf{Id}$-well-foundedness. Recall from the informal introduction of reductivity that it was then expressed in terms of "reductions of the input". The intuition behind reductivity given there boils down to the following. For an $\mathsf{F}$-reductive relation $\mathsf{R}$ we have that whenever $\mathsf{R}$ relates an $\mathsf{F}$-structure $m$ and a point $x$ all the elements of the structure $m$ are related to $x$ by a well-founded relation.

To make this statement precise we need to formalise the concept "membership of $\mathsf{F}$-structures". De Moor and Hoogendijk [43] have shown that this is possible for a certain class of relators. For the relators from that class one can define a membership relation, say $\mathsf{mem}$.

For example, for the list relator this relation holds between a point of the universe and a list precisely when the point is in the list. For product the relation holds between $x$ and $(x,y)$ and also between $y$ and $(x,y)$.

A way of characterising the membership relation of a relator is the following (see [43]):

**Definition 6.46 (Membership)** Relation $\mathsf{mem}$ is a membership relation of relator $\mathsf{F}$ if and only if it satisfies, for all monotypes $A$:

$$\mathsf{F}.A \;=\; \mathsf{F}.\mathrm{I} \circ \mathsf{mem} \backslash A \qquad .$$

□

When this definition is expressed pointwise it reads:

$$x \in \mathsf{F}.A \;\equiv\; x \in \mathsf{F}.\mathrm{I} \wedge \forall(i{:}\, i\langle\mathsf{mem}\rangle x{:}\; i \in A)$$

Or informally: an $\mathsf{F}$-structure is an element of $\mathsf{F}.A$ iff all of its members satisfy property $A$.

A membership relation is a natural transformation of type $\mathsf{F} \overset{\cdot}{\leadsto} \mathsf{Id}$ , and, combining this with corollary 6.15, we deduce that the composition of a membership relation and an $\mathsf{F}$-reductive relation admits induction, or is well-founded.

It is even possible to express reductivity in terms of membership and well-foundedness. Observe that

$$
\begin{array}{ll}
 & \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} \mathsf{F}.A \\
= & \quad \{ \quad \text{membership} \quad \} \\
 & \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} (\mathsf{F}.\mathrm{I} \circ \mathsf{mem} \mathbin{\backslash\!\!\!\downarrow} A) \\
= & \quad \{ \quad \text{monotype factor is conjunctive} \quad \} \\
 & \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} \mathsf{F}.\mathrm{I} \circ \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} (\mathsf{mem} \mathbin{\backslash\!\!\!\downarrow} A) \\
= & \quad \{ \quad \text{monotype factor (2.47)} \quad \} \\
 & \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} \mathsf{F}.\mathrm{I} \circ (\mathsf{mem} \circ \mathsf{R}) \mathbin{\backslash\!\!\!\downarrow} A
\end{array}
$$

If it is now assumed that $\mathsf{R}$ is a coalgebra, i.e. $\mathsf{R}{<} \subseteq \mathsf{F}.\mathrm{I}$ or the equivalent $\mathrm{I} = \mathsf{R} \mathbin{\backslash\!\!\!\downarrow} \mathsf{F}.\mathrm{I}$, the last line reduces to $(\mathsf{mem} \circ \mathsf{R}) \mathbin{\backslash\!\!\!\downarrow} A$. So for coalgebra $\mathsf{R}$ the statement that $\mathsf{R}$ is $\mathsf{F}$-reductive is equivalent to the statement that $\mathsf{mem} \circ \mathsf{R}$ is well-founded.

But all this is only true if the relator has a membership relation. The class of such relators is large. It contains for instance all the polynomial relators, the relators we use in our everyday work. So, although there exist relators that do not have a membership relation it is likely that these are interesting for theoretical reasons only. For a profound discussion of membership the reader is referred to Hoogendijk's forthcoming thesis [54].

The above discussion indicates that it is not absolutely necessary to introduce the notion $\mathsf{F}$-reductivity: one could make do with membership and well-foundedness. This might be the reason why reductivity has not been defined before. However, in our opinion the notion is fundamental. The notion is, for instance, more fundamental than well-foundedness because the standard ways of constructing well-founded relations use initial algebras, or reductive relations. Furthermore, reductivity gives a clearer view of what initiality is all about. One of the main objectives of this chapter was to make this point clear. In the next chapter this matter is discussed further by relating reductivity and termination.

# Chapter 7

# Recursion on Datatypes

## 7.1 Introduction

The central issue of computing science is the development of practical programming methodologies. Such methodologies have been developed, for instance, for the construction of sequential programs and for the construction of functional programs.

An insight that played an important role in the development of a methodology for sequential programs is that it is possible to restrict, without loss of generality, the attention to the class of while programs only. It is not necessary to consider arbitrary goto programs.

In the same spirit, in the methodology for functional programs, arbitrary recursion (which has been called the goto of functional programming) was replaced by structural recursion. However, the problem is that this is a real restriction: not all programs can be expressed using this type of recursion.

One of our goals is the development of a methodology that is an integration and extension of the two methodologies. The class of programs that can be developed using the integrated methodology should not be too restrictive: it should be possible to express any algorithm. Therefore, we can not restrict our attention to structural recursion. On the other hand, arbitrary recursion is too general because allowing it would make it hard to design a methodology, in the same way as it is hard to design a methodology for the construction of programs with goto statements. What we need is a limited form of recursion that is still general enough to express all algorithms.

In this chapter we take a step towards an integrated programming methodology by defining a class of programs with a limited, but not too limited form of recursion. By a program we mean a simple program: one level of recursion and the constituent statements are basic. Think of such a program as a recursive procedure with a body consisting of assignments and if-statements only. This is not a real restriction, because if we know how to construct such simple programs, larger programs can be built by combining them.

This chapter discusses how simple programs can be modelled in a relational framework: as recursive equations in relations. The input-output relation of a program is then a solution of the equation.

A property that a program may or may not have is that it defines the input-output relation, i.e. that it has a unique solution. The property is not without significance because it implies that the program characterises the input-output relation. This means that in proving properties of this input-output relation we can use the program (i.e. the equation) as a characterisation and it is often better to work with characterisations than with a closed formula.

However, from a computational point of view it is yet more important that the recursion of an equation is well-founded. In other words that when the equation is executed as a program all computations will eventually halt. Therefore, we formalise the notion of termination for our class of programs. It will be shown that the fact that a program uniquely determines its input-output relation is a straightforward consequence of termination.

## 7.2 Structural recursion

When we define a relation on the natural numbers (i.e. a relation with right domain the natural numbers and arbitrary left domain) we often do so by structural recursion. That is, the relation is defined by induction on the structure of the natural numbers: we say which values are related to 0 and we say how to form values related to $n+1$ given knowledge about those values related to $n$. That such information *uniquely* characterises the relation we want to define is what makes $\mathsf{zero}{\triangledown}\mathsf{succ}$ an initial algebra. For example, the predicate $\mathsf{even}$ is uniquely characterised by the fact that 0 is even and $n+1$ is not even if and only if $n$ is even.

Formally, we can write the definition of $\mathsf{even}$ as two equations:

$$\mathsf{even} \circ \mathsf{zero} = \mathsf{true} \quad \text{and} \quad \mathsf{even} \circ \mathsf{succ} = \mathsf{not} \circ \mathsf{even} \quad .$$

Better still, using the disjointness of disjoint sum, these two equations can be combined into one:

$$(\mathsf{even}{\circ}\mathsf{zero}) \triangledown (\mathsf{even}{\circ}\mathsf{succ}) \;=\; \mathsf{true} \triangledown (\mathsf{not}{\circ}\mathsf{even}) \quad .$$

"Defusing" this equation by means of $\circ$-$\triangledown$-fusion it the lefthand side and $\triangledown$-$+$-fusion in the righthand side we obtain

$$\mathsf{even} \circ \mathsf{zero}{\triangledown}\mathsf{succ} \;=\; \mathsf{true}{\triangledown}\mathsf{not} \circ \mathbb{1}{+}\mathsf{even} \quad . \tag{7.1}$$

In this form the three components of the equation are readily identified: the relator $(\mathbb{1}+)$, and the two $(\mathbb{1}+)$-algebras $\mathsf{zero}{\triangledown}\mathsf{succ}$ and $\mathsf{true}{\triangledown}\mathsf{not}$.

In the same way, a relation on lists is uniquely defined if we say which elements are related to the empty list and which elements are related to a list that is the "cons" of an element with a list. For example, the relation $\mathsf{prefix}$ is uniquely characterised by the two equations

$$\mathsf{nill}\langle\mathsf{prefix}\rangle\mathsf{nill}$$

(where $\mathsf{nill}$ denotes the empty list) and

$$x\langle\mathsf{prefix}\rangle(\mathsf{cons}.(a,y)) \;\equiv\; x = \mathsf{nill} \vee \exists(z\colon x = \mathsf{cons}.(a,z)\colon z\ \mathsf{prefix}\ y)$$

Expressed as one, point-free equation this is

$$\mathsf{prefix} \circ \mathsf{nil}\triangledown\mathsf{cons} \;=\; \mathsf{nil} \triangledown (\mathsf{nil}\circ\top\!\!\top \cup \mathsf{cons}) \circ \mathbb{1}+(\mathrm{I}\times\mathsf{prefix}) \quad . \tag{7.2}$$

(In order to be formally correct we have used $\mathsf{nill}$ to denote the empty list and $\mathsf{nil}$ to denote the function with right domain $\mathbb{1}$ that returns the value $\mathsf{nill}$.) Again we recognise a relator and two algebras: in this case the relator is $(\mathbb{1}+(\mathrm{I}\times))$ and the two $(\mathbb{1}+(\mathrm{I}\times))$-algebras are $\mathsf{nil}\triangledown\mathsf{cons}$ and $\mathsf{nil} \triangledown (\mathsf{nil}\circ\top\!\!\top \cup \mathsf{cons})$. That equations like (7.2) uniquely define a relation is what makes $\mathsf{nil}\triangledown\mathsf{cons}$ an initial $(\mathbb{1}+(\mathrm{I}\times))$-algebra.

The two equations (7.1) and (7.2) are examples of the general type of recursive definition

$$\mathsf{X} \circ \mathsf{in} \;=\; \mathsf{R} \circ \mathsf{F.X} \tag{7.3}$$

where $\mathsf{in}$ is an initial $\mathsf{F}$-algebra. This type of definition is used in modern functional programming languages like Haskell. Such a definition is called "a definition by pattern matching": the effect of the program depends on the pattern of the input. The pattern of the input is, in turn, determined by the constructor of the initial algebra that is applied to construct the input. For instance, for lists the two possible input patterns are the empty and non-empty lists, respectively.

Because equation (7.3) is the abstract version of a functional program it can indeed be considered as a program. That is why we use the terms programs and equations interchangeably for this type of equations. It should be noted that it is implicit in definitions (7.1) and (7.2) that $\mathsf{even}$ and $\mathsf{prefix}$ are only defined on natural numbers and lists, respectively. To make this assumption explicit, which we have to do in our untyped system, we add to (7.3) the requirement that the right domain of $\mathsf{X}$ is contained in the carrier of the initial algebra.

$$\mathsf{X} = \mathsf{X} \circ \mathsf{in}_{<} \tag{7.4}$$

According to the definition of initial algebra (6.27) the conjunction of (7.3) and (7.4) has a solution (in fact a unique one). However, the equation is not easy to generalise. If the initial algebra is replaced by an arbitrary algebra, even the existence of a solution is no longer guaranteed. The reason is that the equation is not a fixed point equation and it is therefore not possible to apply the Knaster-Tarski fixed point theorem to show that a solution exists.

To obtain an equation that is a fixed point equation of a monotone function we shift the initial algebra to the right side of the equality sign. This is possible because of (6.31). We so obtain the *cata equation*

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \circ \mathsf{F.X} \circ \mathsf{in}^{\cup} \quad . \tag{7.5}$$

The unique solution of the cata equation is called a *catamorphism*.

A cata equation can also be seen as a program. For instance, when we instantiate it for the $\mathsf{prefix}$ example the equation becomes

$$\mathsf{prefix} \;=\; \mathsf{nil} \triangledown (\mathsf{nil}\circ\top\!\!\top \cup \mathsf{cons}) \circ \mathbb{1}+(\mathrm{I}\times\mathsf{prefix}) \circ \mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup} \quad . \tag{7.6}$$

(Recall that the junc and cojunc operators are each other's converse dual, therefore we have $(\mathsf{nil} \triangledown \mathsf{cons})^{\cup} = \mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup}$).

In chapter 2 it was discussed how programs like (7.6) can be interpreted. The righthand side can be seen as a selection statement: a test is made to determine whether or not the input is the empty list. If the input list, $m$ say, is not empty, $m$ is decomposed into its head and tail part by $\mathsf{cons}^{\cup}$, next $\mathsf{prefix}$ is called recursively on the tail of $m$. The last statement $\mathsf{nil}\circ\top\!\!\top \cup \mathsf{cons}$ of this branch is a non-deterministic choice: the output of the program is either the empty list or the list with the head of $m$ as head and the result of the recursive call as tail. The other branch of the selection statement is executed if the input is the empty list. The output is then also the empty list.

Note that, whereas (7.3) corresponds to the way programs are written in functional languages, the shape of program (7.5) is close to the form of recursive procedures in Pascal. The right side is the body of the recursive procedure.

Because equations of the form (7.5) can be considered as programs, we call such equations cata programs. The unique solution of a cata program can be considered to be the input-output relation of the program and is also called "the relation computed by the program".

Structural recursion has an important application: it can be used to define new relators, the so-called *map*-relators. Recall that relators are special functions from relations to relations. Given a relation $\mathsf{R}$ the relation $\mathsf{F.R}$ is to be interpreted as the relation "$\mathsf{R}$ applied element-wise". More precisely it is a relation between $\mathsf{F}$-structures such that elements of the input structure are related to corresponding elements of the output structure by relation $\mathsf{R}$.

In the concrete case of the list relator this means that $\mathsf{List.R}$ is a relation between lists of equal length such that the corresponding elements of the lists are related by $\mathsf{R}$. Consequently, $\mathsf{List.R}$ relates the empty list to the empty list. For a non-empty list we have

$$\mathsf{cons}.(a,y)\langle\mathsf{List.R}\rangle\mathsf{cons}.(b,m) \;\equiv\; a\langle\mathsf{R}\rangle b \wedge y\langle\mathsf{List.R}\rangle m \quad ,$$

which is: the heads of the lists are related by $\mathsf{R}$, the tails by $\mathsf{list.R}$. This can be written point-free as

$$\mathsf{List.R} \;=\; \mathsf{nil}\triangledown\mathsf{cons} \circ \mathbb{1}+(\mathsf{R}\times\mathrm{I}) \circ \mathbb{1}+(\mathrm{I}\times\mathsf{List.R}) \circ \mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup} \tag{7.7}$$

We can abstract from this particular example to obtain the definition of "map" relators. Define the binary relator $\otimes$ by

$$\mathsf{R}\otimes\mathsf{S} = \mathbb{1}+(\mathsf{R}\times\mathsf{S})$$

With this definition the expressions $\mathbb{1}+(\mathsf{R}\times\mathrm{I})$ and $\mathbb{1}+(\mathrm{I}\times\mathsf{list.R})$ can be rewritten as $\mathsf{R}\otimes\mathrm{I}$ and $\mathrm{I}\otimes\mathsf{list.R}$, respectively. Furthermore we have that $\mathsf{nil}\triangledown\mathsf{cons}$ is an initial $(\mathrm{I}\otimes)$-algebra.

To abstract from the structure of this algebra we call it $\varphi$. Now we can rewrite (7.7) and define the function $\mathsf{list}$ as

$$\mathsf{list}.\mathsf{R} \;=\; \varphi \circ \mathsf{R}{\otimes}\mathrm{I} \circ \mathrm{I}{\otimes}\mathsf{list}.\mathsf{R} \circ \varphi^{\cup} \tag{7.8}$$

Of course, this type of definition works for any binary relator. Let $\oplus$ be such a binary relator and let $\psi$ be an arbitrary initial $(\mathrm{I}\oplus)$-algebra (it is not difficult to verify that $(\mathrm{I}\oplus)$ is a relator). Then the function $\mathsf{map}$ defined by

$$\mathsf{map}.\mathsf{R} \;=\; \psi \circ \mathsf{R}{\oplus}\mathrm{I} \circ \mathrm{I}{\oplus}\mathsf{map}.\mathsf{R} \circ \psi^{\cup} \tag{7.9}$$

is a relator. To prove this one only needs to know that $\mathsf{map}.\mathsf{R}$, being a unique solution, is a least fixed point. So, by monotonicity of the least fixed point operator, $\mathsf{map}$ is monotone. That $\mathsf{map}$ commutes with the converse operator is straightforward: it follows because $\oplus$ and converse commute. An appeal to the Knaster-Tarski theorem is needed to show that $\mathsf{map}.\mathrm{I} \subseteq \mathrm{I}$; the resulting expression $\psi \circ \mathrm{I}{\oplus}\mathrm{I} \circ \mathrm{I}{\oplus}\mathrm{I} \circ \psi^{\cup} \subseteq \mathrm{I}$ is true because $\mathrm{I}{\oplus}\mathrm{I}$ is a monotype and the initial algebra is functional. The final thing to show, viz. $\mathsf{map}$ distributes over composition is, of course, shown by applying $\mu$-fusion; one also needs that $\oplus$ distributes over composition and that $\psi \circ \psi^{\cup} = \mathrm{I} \oplus \psi^{<}$.

If we take the special initial algebra $\mu(\mathrm{I}\oplus)$ for $\psi$, it can be shown that, with that choice, $\mathsf{map}.\mathsf{R}$ is equal to $\mu(\mathsf{R}\oplus)$. See [8] for more details.

## 7.3 Primitive recursion

Structural recursion is useful: many programs that arise in practice have this kind of recursion and it can also be used to define new relators. However, just as structural induction is not enough to prove all facts that can be proved by induction, structural recursion is not enough to define all programs that can be defined by recursion. As an example of a program that is not structural recursive, consider the factorial function, the function defined by the two equations

$$\mathsf{fact} \circ \mathsf{zero} = \mathsf{one} \quad \text{and} \quad \mathsf{fact} \circ \mathsf{succ} = \mathsf{times} \circ \mathsf{fact}{\vartriangle}\mathsf{succ} \;\;,$$

where $\mathsf{one}$ is the constant function returning the number 1 and $\mathsf{times}$ is the multiplication function. These equations can be combined into the single equation

$$\mathsf{fact} \;=\; \mathsf{one}{\triangledown}(\mathsf{times} \circ \mathsf{succ}{\times}\mathrm{I}) \circ \mathbb{1}{+}(\mathrm{I}{\vartriangle}\mathsf{fact}) \circ \mathsf{zero}^{\cup} \blacktriangledown \mathsf{succ}^{\cup} \quad . \tag{7.10}$$

The presence of the split operator $(\vartriangle)$ in this definition is because of the fact that the result of the factorial function depends, in addition to the result of the recursive call, directly on the input: $(n+1)! = n! * (n+1)$, the input $n$ appears twice in the right side. This means that the input $n$ is copied. If one gives a pointwise definition this copying is done by writing $n$ twice. However, in a point-free formulation the copying has to be coded by a relation, in this case by the doubling function $\mathrm{I}{\vartriangle}\mathrm{I}$.

Instances of the doubling and similar functions occur when a point-wise formula is rewritten in a point-free form. The location and number of occurrences of the points in the point-wise formula have to be coded by a relation in the point-free formula. Such relations are called (somewhat derogatively) *plumbing* relations.

Because of the presence of a plumbing relation the form of equation (7.10) differs from the form of the cata equation (7.5) and the program is therefore not structurally recursive. The question is whether equation (7.10) is a definition, i.e. is there a unique solution? If not, we are obliged to define $\mathsf{fact}$ by a phrase like: "the factorial function is the least solution of the equation (7.10)". Of course it is well-known that this is unnecessary: equation (7.10) has a unique solution and characterises the factorial function. Although this fact is a consequence of the definition of initial algebra it is not an immediate consequence and, therefore, requires a proof. (We present a proof later.)

When we abstract from the particular relator and initial algebra in factorial program (7.10) a general recursion scheme is obtained.

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \circ \mathsf{F}.(\mathrm{I}\times\mathsf{X}) \circ \mathsf{F}.(\mathrm{I}\vartriangle\mathrm{I}) \circ \mathsf{in}^{\cup} \quad . \tag{7.11}$$

(Note that we have made the plumbing relation explicit by means of $\times$-$\vartriangle$-fusion.) An equation of this form is called a *para equation*; the least solution of a para equation, (which is also the unique solution), is called a *paramorphism*. Alternatively, we can say: programs with this kind of recursion are para programs, the relation computed by such a program is a paramorphism. So, (7.10) is a para program to compute the factorial function (which, therefore, is a paramorphism).

Para equations and paramorphisms were studied by Meertens [68] and he has shown that the unique solution of a para equation can be expressed as the composition of a projection function and the unique solution of a cata equation. This result can also be interpreted as: any paramorphism can be computed from the output of a certain cata program. However, one can not conclude that every paramorphism *is* a catamorphism: it is not true that every para program can be transformed into a cata program that computes the same relation.

However, it is the case that any catamorphism is also a paramorphism: any catamorphism can be computed by a para program. This is because a cata equation is a special case of a para equation. In other words any cata program can be transformed into a para program. To see this, substitute $\mathsf{R} \circ \mathsf{F}.\gg$ for $\mathsf{R}$ in (7.11). As a consequence of the rule $\gg \circ\, \mathrm{I}\vartriangle\mathsf{X} = \mathsf{X}$ the equation then reduces to a cata equation.

So, the type of recursion of the para equation is more general than structural recursion. The recursion pattern of the para equation is known as *primitive* recursion. Because primitive recursion is more general than structural recursion one might be tempted to use primitive recursion rather than structural recursion as the standard recursion scheme. This is indeed done in, for example, Martin-Löf's theory of types.

However, as the name suggests, primitive recursion is still not general enough to capture all useful recursion schemes. Not all programs we want to consider are para programs.

## 7.4 Hylo programs

The observation that not all programs are primitive recursive is of course not new. For instance in the well-known definition of the computable functions on the natural numbers the primitive recursive functions are a proper subclass of the class of all computable functions. Therefore, a number of researchers have proposed generalisations of the para equation. Malcolm [62] defined the *zygo* equation, which type of equation was generalised by Fokkinga [42] to the *mutu* equation

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \circ \mathsf{F.X} \vartriangle \mathsf{F.X} \circ \mathsf{in}^{\cup} \quad . \tag{7.12}$$

The equation is called a mutu equation because such equations arise if one wants to formalise mutually recursive programs. It can be shown that such an equation has a unique solution. (However, Fokkinga's proof requires that relation $\mathsf{R}$ is a functional and total algebra. So, if we want to show that equation (7.12) also has a unique solution for an arbitrary relation $\mathsf{R}$ we can not reuse Fokkinga's proof.)

The problem with this kind of generalisation of primitive recursion is that it is rather limited. The generalised programs are of the form

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \circ \mathsf{G.X} \circ \mathsf{plumbing} \circ \mathsf{in}^{\cup} \quad , \tag{7.13}$$

where $\mathsf{G}$ is a relator, $\mathsf{in}$ an initial $\mathsf{F}$-algebra for some relator $\mathsf{F}$, and $\mathsf{plumbing}$ a plumbing relation. Recall that a plumbing relation is only a coding of "where the arguments have to go", so it does not represent a proper computation. Yet, many programs perform a substantial amount of preprocessing before the recursive call and are therefore not of the form (7.13).

Consider, for instance, the program known as "quicksort" .

$$\mathsf{qs} \;=\; \mathsf{nil} \triangledown (\mathsf{join} \circ \mathrm{I} \times \mathsf{cons}) \circ \mathbb{1} + (\mathsf{qs} \times (\mathrm{I} \times \mathsf{qs})) \circ \mathsf{nil}^{\cup} \blacktriangledown (\mathsf{dnf} \circ \mathsf{cons}^{<}) \tag{7.14}$$

To see that this is the quicksort program one has to interpret $\mathsf{dnf}$ as the well-known "Dutch national flag" operation. This operation splits a non-empty list into a tuple $(m,(a,n))$ formed by a list, an element and a list such that all elements in the list $m$ are at most $a$ and all elements in $n$ are greater than $a$. The results of the recursive calls are assembled to the output list by the operation $\mathsf{join} \circ \mathrm{I} \times \mathsf{cons}$, where $\mathsf{join}$ produces the concatenation of two lists.

The recursion of the quicksort program differs from the recursion of equation (7.13) because relation $\mathsf{dnf}$ can not be considered as a simple plumbing relation: it is not merely a copying and rearranging of the input values in the sense that it does not disappear if formula (7.14) is expressed point-wise. In short, $\mathsf{dnf}$ represents a considerable amount of preprocessing.

Therefore, we might want to consider equations of the form

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R} \circ \mathsf{G.X} \circ \mathsf{prepro} \circ \mathsf{in}^{\cup} \quad , \tag{7.15}$$

as the standard recursion scheme. Fokkinga has studied a limited class of equations of this form (the class was limited because there were a number of restrictions on the preprocessor relation $\mathsf{prepro}$ and relator $\mathsf{G}$). He has shown that such equations have a unique solution and called that unique solution a *prepromorphism*.

However, for our purposes (7.15) is still not general enough. We want for instance to capture divide and conquer algorithms and such programs do not, in general, have the form of equation (7.15) because no initial algebra is involved. A typical divide and conquer program is of the form

$$\mathsf{X} \;::\; \mathsf{X} = \mathsf{R}\triangledown\mathsf{conquer} \circ \mathrm{I}{+}\mathsf{X}{\times}\mathsf{X} \circ \mathrm{I}{+}\mathsf{divide} \circ \mathsf{A}_\blacktriangledown\mathsf{B} \quad . \tag{7.16}$$

Interpreting this program should not be difficult. A test is made to determine whether the input is a base case (if the input satisfies $\mathsf{A}$), the output then being computed by $\mathsf{R}$. If the input is not a base case (if the input satisfies $\mathsf{B}$) the input is split into two smaller "subproblems" by $\mathsf{divide}$. Then the smaller problems are solved recursively and finally the two solutions of the subproblems are assembled into an output by $\mathsf{conquer}$.

Of course there are more divide and conquer programs: e.g. the original problem can be split into more than two subproblems. It is also possible that the divide step produces, besides a number of subproblems, a value that is not "passed into the recursion" ; then the middle relation of (7.16) has a form like $\mathrm{I}{\times}(\mathsf{X}{\times}\mathsf{X})$. Quicksort is an example of such a divide and conquer algorithm.

Now it is time to define the type of programs we propose as the standard recursion scheme: it is a program like the cata program but we do not require that the preprocessor is the converse of an initial algebra. Such programs are called *hylo* programs.

**Definition 7.17 (Hylos)** Let $\mathsf{R}$ and $\mathsf{S}$ be relations and $\mathsf{F}$ a relator. An equation of the form

$$\mathsf{X} :: \mathsf{X} = \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R} \tag{7.18}$$

is said to be a *hylo equation* or *hylo program*.

The least solution of equation (7.18) is called a *hylomorphism* and is denoted by $[\![\mathsf{F};\mathsf{S},\mathsf{R}]\!]$. (If the relator is understood from the context the least solution is denoted by $[\![\mathsf{S},\mathsf{R}]\!]$.)
□

(The name hylomorphism for the least solution of a hylo equation appeared in [42]. )

The hylo equation generalises all the defining equations of the "morphisms" we have encountered thus far. However, it generalises more, for instance the *ana equation* [69] mentioned here because it is needed later on.

$$\mathsf{X} :: \mathsf{X} = \mathsf{in} \circ \mathsf{F.X} \circ \mathsf{R} \tag{7.19}$$

(Ana equations have a unique solution called *anamorphisms*.)

Another, perhaps more familiar, program that has the form of a hylo equation is the repetition. The repetition can be written recursively as:

$$x \quad = \quad \mathsf{if}\ \neg \mathsf{b} \rightarrow \mathsf{skip}\ [\!]\ \mathsf{b} \rightarrow \mathsf{s};x\ \mathsf{fi} \quad . \tag{7.20}$$

Using the fact that skip (do nothing) corresponds to the relation I and writing B for the monotype corresponding to predicate b, we may express (7.20) using disjoint sum as:

$$\mathsf{X} \quad = \quad \mathrm{I} \triangledown \mathrm{I} \circ \mathrm{I}{+}\mathsf{X} \circ {\sim}\mathsf{B} \blacktriangledown (\mathsf{S}{\circ}\mathsf{B}) \tag{7.21}$$

Thus a repetition is a hylo equation.

Recall that one of the goals of this chapter was to identify a recursion scheme that makes arbitrary recursion superfluous. The question is now whether the class of hylo programs forms a good candidate for such a recursion scheme. Specifically, can everything that can be computed also be computed by a hylo program?

It is well-known that everything that can be computed can be computed with a repetition. Now a repetition is a hylo program. Therefore, the class of hylo equations is large enough to write all programs we might want to: every program can be transformed into a hylo program. In other words we can safely restrict the class of programs we want to consider to the class of hylo programs.

The class of programs can also be safely restricted to just repetitions. However, although that is safe it would not be convenient because there are many practical algorithms which can not be expressed as repetitions in a simple way (quicksort is an example). It would not be convenient because we are then forced to express such programs as repetitions, thereby making them unnecessarily complicated.

The question now is whether it is convenient to require that all algorithms are expressed in the form of hylo programs. The answer seems to be positive because the great majority of programs that one encounters in practice are in the form of a hylo equation. The few programs that are not are by no means standard, much used algorithms. On the contrary: they tend to be rather artificial. An example is, for instance, the program to compute the Ackerman function.

Because of these considerations we conclude that the recursion scheme of the hylo program is sufficiently general that it is both possible in theory and convenient in practice to restrict our attention to hylo programs only. In other words, we suggest that a useful method of programming can be developed for the construction of hylo programs. This method should extend the method of developing repetitions by means of invariants, guards and bound functions and should have the same spirit.

Such a method gives greater freedom in designing programs because one is not forced to apply the solution strategy of using a repetition. Therefore, one of the ingredients of the method should be heuristics suggesting suitable solution strategies. Now the relator in the hylo program can be seen as a representation of such a strategy. For instance relator $\mathsf{X} \mapsto \mathrm{I}{+}\mathsf{X}$ is connected to the repetition, $\mathsf{X} \mapsto \mathrm{I}{+}\mathsf{X}{\times}\mathsf{X}$ represents a divide and conquer strategy, and $\mathsf{X} \mapsto \mathsf{F}.(\mathrm{I}{\times}\mathsf{X})$ corresponds to primitive recursion. This gives a procedure to determine the relator of a hylo program under construction: as soon as the decision has been taken which solution strategy has to be applied the overall shape of the relator is known.

Before we can begin to develop a method of designing hylo programs, there are some other problems left to solve. One of these is that typically equation (7.18) will not have a unique solution (e.g. take for $\mathsf{F}$ the identity relator and for $\mathsf{R}$ and $\mathsf{S}$ the identity relation). From a programmer's point of view this may not seem much of a problem: his concern is whether or not a program terminates. However, the researchers working on the "morphisms" had more than just mathematical reasons to stress that their equations have a unique solution. There is a reason why unicity of solution is of interest for computing scientists: it is known (although the fact is seldom formulated) that there is some connection between the existence of a unique solution and well-foundedness of the recursion. In the next section we give a condition which guarantees that a hylo program terminates and show that this condition also implies that the program has a unique solution.

## 7.5 Reductivity and termination

In order to derive a definition of termination, we must first have an idea of how a hylo program,

$$\mathsf{X} = \mathsf{S} \circ \mathsf{F}.\mathsf{X} \circ \mathsf{R}$$

is to be executed. As long as no non-determinism is involved there seems to be only one reasonable choice: the program is executed by first unfolding the equation and then computing the argument for the recursive call by executing $\mathsf{R}$. This procedure is repeated until a base case is reached and no further unfoldings are necessary. Then the output is computed by executing $\mathsf{S}$ as often as the equation was unfolded. If no base case is reached then the execution does not terminate and no output is produced. Somewhat more precisely: for each input which gives rise to a terminating computation there exists a natural number $n$ such that the output value is related to the input value by relation

$$\mathsf{S} \circ \mathsf{F}.\mathsf{S} \circ \cdots \circ \mathsf{F}^n.\mathsf{S} \circ \mathsf{F}^{n+1}.\perp\!\!\perp \circ \mathsf{F}^n.\mathsf{R} \circ \cdots \circ \mathsf{F}.\mathsf{R} \circ \mathsf{R} \tag{7.22}$$

(where the relation $\mathsf{F}.\perp\!\!\perp$ is the input-output relation for the base case). This is the familiar execution scheme applied by the implementations of imperative languages like Pascal and functional languages such as Haskell. (Because of this execution scheme, the computed input-output relation is the least solution of the hylo program. Relators that arise in practice are continuous, so the union over all $n$ of the relations (7.22) is the least solution of the program.)

If $\mathsf{R}$ is deterministic there is only one possible argument for each recursive call (and therefore only one value for $n$). However if $\mathsf{R}$ is non-deterministic then each unfolding can give rise to many possibilities for continuing the computation. The question is now which of those possible continuations is chosen. The standard semantics of the guarded command language is defined in such a way that one of the possibilities can be chosen non-deterministically. However, there is of course another scenario imaginable: all possibilities could be pursued, maybe in parallel, until a base case is reached.

An ideal implementation of Prolog should do this. There is then the problem that $\mathsf{R}$ can be unboundedly nondeterministic: there can be an infinite number of possible continuations. In that case (assuming that there exist no machines with unbounded parallellism) there can be two causes of non-termination: either there is never a base case reached or at some point infinitely many possible continuations have to be followed. However, if $\mathsf{R}$ is boundedly non-deterministic this execution scheme will more often lead to a terminating computation than the execution scheme of the guarded command language.

For a given input, exploring all possible continuations in parallel will result in a terminating computation if one of the possible continuations leads to a base case. On the other hand, in the case that one continuation is chosen non-deterministically, termination can only be guaranteed if all continuations lead to a base case.

Although the execution scheme of the guarded command language is perhaps less attractive because it terminates less often we choose that as the way the hylo program is run. The reason is that choosing a continuation non-deterministically, rather than pursuing all possible continations, is compatible with the standard implementations of functional and imperative languages.

To obtain a definition of termination for a hylo program we use the same approach as we did in chapter 5 for the repetition. Given program

$$\mathsf{X} = \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R} \quad , \tag{7.23}$$

a set of states is called "safe" if the program is guaranteed to terminate when started in a state from the set. The union of all safe sets is called the maximal safe set and is modelled by monotype $\mathsf{B}$. So, maximal safe set $\mathsf{B}$ is to be interpreted as the set of all states in which the program can be started without the risk of non-termination. Note that, with this interpretation of safe, a set is safe if and only if it is contained in $\mathsf{B}$.

The assumption is now that the maximal safe set exists and our goal is to characterise this set.

A first requirement on a safe set $\mathsf{A}$ (i.e. a set with property $\mathsf{A} \subseteq \mathsf{B}$) is that, whenever program $\mathsf{X}$ is started in it, all possible recursive calls terminate: we can not be sure which of them is chosen. So, if the program is started in set $\mathsf{A}$, the recursive call should be started in the maximal safe set $\mathsf{B}$. This can be expressed as:

$$\mathsf{X} \circ \mathsf{A} \subseteq \mathsf{S} \circ \mathsf{F}.(\mathsf{X} \circ \mathsf{B}) \circ \mathsf{R} \circ \mathsf{A} \quad . \tag{7.24}$$

Because relators distribute over composition and composition is monotone,this boils down to requiring property $(\mathsf{R} \circ \mathsf{A})< \subseteq \mathsf{F.B}$ to hold:

$$
\begin{array}{ll}
 & \mathsf{S} \circ \mathsf{F}.(\mathsf{X} \circ \mathsf{B}) \circ \mathsf{R} \circ \mathsf{A} \\
= & \quad \{ \text{ relators distribute over composition } \} \\
 & \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{F.B} \circ \mathsf{R} \circ \mathsf{A} \\
= & \quad \{ \text{ assumption: } (\mathsf{R} \circ \mathsf{A})< \subseteq \mathsf{F.B} \\
 & \quad\;\; \text{ domains: } \mathsf{Y}< \subseteq \mathsf{C} \Rightarrow \mathsf{C} \circ \mathsf{Y} = \mathsf{Y} \; \} \\
 & \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R} \circ \mathsf{A} \\
= & \quad \{ \; \mathsf{X} \text{ solves (7.23) } \} \\
 & \mathsf{X} \circ \mathsf{A}
\end{array}
$$

The requirement on $A$ can thus be formalised as:

$$A \subseteq B \quad \Rightarrow \quad (R \circ A)_< \subseteq F.B \tag{7.25}$$

Conversely, if property $(R \circ A)_< \subseteq F.B$ holds (i.e. the recursive calls are started in the safe set $B$ whenever the program is started in set $A$) then set $A$ can be considered safe and should therefore be contained in the maximal safe set:

$$A \subseteq B \quad \Leftarrow \quad (R \circ A)_< \subseteq F.B \tag{7.26}$$

Now we can make use of the defining Galois connection of the monotype factor: expression $(R \circ A)_<$ can be rewritten as $A \subseteq R \backslash F.B$. Therefore (7.25) and (7.26) can be combined:

$$\forall (A :: \ A \subseteq B \equiv A \subseteq R \backslash F.B) \tag{7.27}$$

The rule of indirect equality gives that (7.27) is equivalent to $B = R \backslash F.B$. This leaves us with the choice which solution of this equation in $B$ to take. Basically there are only two obvious choices: the least and the largest solution. In general, however, the largest solution is equal to the identity relation (for $R$ a coalgebra, which is a reasonable assumption on typing grounds). So if we take that one as a safe set we adopt the view that the computation can be started safely in any state, regardless of what $R$ is. This would obviously be too optimistic. Therefore we take the least solution as the maximal safe set. (The pessimistic view is a "demonic" view as opposed to an "angelic" view.)

**Definition 7.28 (Safe set)** The safe set of program $X = S \circ F.X \circ R$ is the monotype $\mu(A \mapsto R \backslash F.A)$.
□

Now the execution of program $X = S \circ F.X \circ R$ terminates if the program is started in its safe set. Admittedly, this definition is a bit informal: we did not define what it means for a program (an equation) to be started in a monotype. This can of course be made precise, but we will not do so here because in this chapter we are interested only in programs which terminate everywhere. Such a program will be called a terminating program and the notion "terminating program" can be defined (precisely) in terms of reductivity.

If $\mu(A \mapsto R \backslash F.A) = I$ , the program can be started safely in any state. Conversely, if $\mu(A \mapsto R \backslash F.A) \neq I$ there is a non-safe state because $\mu(A \mapsto R \backslash F.A)$ is the maximal safe set. This gives the connection between reductivity and termination.

**Definition 7.29 (Terminating program)** Program $X = S \circ F.X \circ R$ is a terminating program if and only if the relation $R$ is $F$-reductive.
□

Note that reductivity states that the equation $B = R \backslash F.B$ has only one solution. So if we restrict our attention to terminating programs in the sense of definition 7.29 the choice which solution of this equation should be taken as the maximal safe set becomes irrelevant.

Having defined termination, we have to check that the definition given here is compatible with the programmer's definition. A programmer proves termination by using well-founded relations: he has to prove that the argument of every recursive call is "smaller" than the original argument. For hylo program $\mathsf{X} = \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R}$ this means that all members of an output $\mathsf{F}$-structure of $\mathsf{R}$ have to be smaller than the corresponding input of $\mathsf{R}$. More formally, with $x\langle\mathsf{mem}\rangle y$ standing for "$x$ is a member of $\mathsf{F}$-structure $y$", we need for all $x$ and $z$

$$\forall(y :: \; x\langle\mathsf{mem}\rangle y \wedge y\langle\mathsf{R}\rangle z \Rightarrow x \prec z) \quad ,$$

for some well-founded ordering $\prec$. This condition can be rewritten as $\mathsf{mem} \circ \mathsf{R} \subseteq \prec$, for some well-founded $\prec$, i.e. as $\mathsf{mem} \circ \mathsf{R}$ is well-founded. This, however, means that the old and the new definition are compatible, because, as we have seen in the previous chapter, relation $\mathsf{mem} \circ \mathsf{R}$ is well-founded iff $\mathsf{R}$ is ($\mathsf{F}$-)reductive. (Of course this only works if $\mathsf{F}$ has a membership relation. Defining termination in terms of reductivity instead of well-foundedness and membership has the advantage that termination can be defined without any restriction on $\mathsf{F}$.)

Because an $\mathsf{F}$-reductive relation is also $\mathsf{F}$-well-founded we have that a consequence of termination is that a terminating program has a unique solution (i.e. a unique input-output relation).

**Lemma 7.30** If program $\mathsf{X} = \mathsf{S} \circ \mathsf{F.X} \circ \mathsf{R}$ is terminating it has a unique solution.

**Proof** Definition 7.29, theorem 6.25 and definition 6.24.
□

We conclude this section by sketching how an alternative definition of termination can be obtained. Recall that definition 7.29 of termination depends on the assumption that if there is, due to non-determinism, at a certain point during the execution more than one possibility to proceed, only one of those possibilities is chosen. Had we adopted the other assumption, viz. that all possible continuations of the executions are pursued, it would have turned out that the maximal safe set should be a solution of the equation $\mathsf{B} = (\mathsf{F.B} \circ \mathsf{R})_{<}$. This equation is kind of dual to the equation that was used to obtain definition 7.29 (in fact it is dual by replacing the monotype factor by its lower adjoint). Termination as defined in this section has a close connection to reductivity, the alternative definition seems to have an equally close connection to inductivity.

## 7.6 Proofs of termination: reductivity arguments

In this section we apply the techniques developed thus far to give proofs of termination of a number of abstract and concrete programs. Because termination is defined in terms of reductivity, such proofs are reductivity arguments. First we tackle a few simple cases: cata, para and mutu programs.

The cata program

$$\mathsf{X} \ :: \ \mathsf{X} = \mathsf{R} \circ \mathsf{F.X} \circ \mathsf{in}^{\cup}$$

is according to definition 7.29 terminating because (theorem 6.44) the converse of an initial F-algebra is F-reductive.

Termination of the para program

$$\mathsf{X} \ :: \ \mathsf{X} = \mathsf{R} \circ \mathsf{F}.(\mathrm{I}{\times}\mathsf{X}) \circ \mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I}) \circ \mathsf{in}^{\cup}$$

is also straightforward to show. Relation $\mathsf{in}^{\cup}$ is F-reductive. Furthermore, we have

$$\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I}) \circ \mathsf{F.A} \subseteq \mathsf{F}.(\mathrm{I}{\times}\mathsf{A}) \circ \mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})$$

To show this one needs that relators distribute over composition, that composition distributes over split (i.e. $\mathrm{I}{\vartriangle}\mathrm{I} \circ \mathsf{Y} \subseteq \mathsf{Y}{\vartriangle}\mathsf{Y}$), the rule $\times$-$\vartriangle$-fusion, that $\mathsf{A}$ is a monotype and also that relators are monotone. This means that relation $\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I})$ is a natural transformation of type $\mathsf{F} \stackrel{\cdot}{\leadsto} \mathsf{F}{\circ}(\mathrm{I}{\times})$. It now follows by corollary 6.15 that $\mathsf{F}.(\mathrm{I}{\vartriangle}\mathrm{I}) \circ \mathsf{in}^{\cup}$ is an $(\mathsf{F}{\bullet}(\mathrm{I}{\times}))$-reductive relation. Hence, by definition 7.29 the para program is terminating (and has, therefore a unique solution).

The proof that a mutu program

$$\mathsf{X} \ :: \ \mathsf{X} \ = \ \mathsf{R} \circ \mathsf{F.X}{\vartriangle}\mathsf{F.X} \circ \mathsf{in}^{\cup}$$

is terminating is similar to the proof for the para program. One needs to check that $\mathrm{I}{\vartriangle}\mathrm{I}$ is of type $\mathsf{F} \stackrel{\cdot}{\leadsto} \mathsf{G}$, where relator $\mathsf{G}$ is defined by $\mathsf{G.Y} = \mathsf{F.Y} \times \mathsf{F.Y}$, i.e. one has to show that

$$\mathrm{I}{\vartriangle}\mathrm{I} \circ \mathsf{F.A} \ \subseteq \ \mathrm{I}{\vartriangle}\mathrm{I} \ \circ \ \mathsf{F.A} \times \mathsf{F.A}$$

which follows immediately by distribution of composition over split and $\times$-$\vartriangle$-fusion.

To give a somewhat more involved example we consider a program that has additional parameters. The program has the form

$$\mathsf{X} \ = \ \mathsf{R} \circ \mathsf{F.X} \times \mathrm{I} \circ \mathsf{F}.(\mathrm{I}{\times}\mathsf{S}) \times \mathrm{I} \circ \mathsf{pass} \vartriangle \gg \circ \ \mathsf{in}^{\cup} \times \mathrm{I} \tag{7.31}$$

What we need for proving termination of program (7.31) is that relation $\mathsf{pass}$ has the property that for all monotypes $\mathsf{A}$

$$\mathsf{pass} \circ \mathsf{F.A} \times \mathsf{B} \ \subseteq \ \mathsf{F}.(\mathsf{A}{\times}\mathsf{B}) \circ \mathsf{pass} \quad . \tag{7.32}$$

It can be shown that for any relator built from product, sum and map relators such a relation $\mathsf{pass}$ can be constructed. The importance of such plumbing relations is that they can be used to express that a parameter is "passed into the recursion".

Program (7.31) is not easy to interpret. Fortunately, it is also not necessary to interpret the program to give the termination proof. However, if one insists on interpreting this could be done in the following way. To make the interpretation easier we take a concrete example: let $\mathsf{F}$ be the relator $(\mathbb{1}+)$ and $\mathsf{in}$ the initial algebra $\mathsf{zero}{\triangledown}\mathsf{succ}$. The reader is warned that it is, even with this concrete example, difficult. It needs quite some practice to master the interpretation of such point-free programs.

The input of the program is a pair $(n, p)$ consisting of a natural number and a parameter. Relation $\mathsf{in}^{\cup} \times I$ takes this input into an element of $F.\mathsf{in}_{<} \times I$. Depending on whether or not $n$ is 0 such an element has either the form $(\hookrightarrow.*,p)$ or $(\hookleftarrow.(n-1),p)$. Then this pair is doubled and the left pair of that doubled pair is transformed by $\mathsf{pass}$ into either $\hookrightarrow.*$ or $\hookleftarrow.(n-1,p)$, whereas the right pair of the doubled pair is reduced by the projection function to $p$. The new intermediate result is thus $(\hookrightarrow.*,p)$ or $(\hookleftarrow.(n-1,p),p)$. Relation $F.(I{\times}S)\times I$ transforms this into either $(\hookrightarrow.*,p)$ or $(\hookleftarrow.(n-1,q),p)$. If the intermediate result is now of the form $(\hookrightarrow.*,p)$ the base case has been reached and the argument $(\hookrightarrow.*,p)$ is passed to $R$, otherwise the program is recursively applied to the component $(n-1,q)$ of $(\hookleftarrow.(n-1,q),p)$, the result of the recursive call being $r$ say. In this case $(\hookleftarrow.r,p)$ is passed to $R$.

It is a lot simpler to express the effect of the program point-wise. However, this only works for functional $R$ and $S$, where $R$ is of the form $T \triangledown U$.

$$X.(0,p) = T.p \quad \text{and} \quad X.(n+1,p) = U.(X.(n, S.p),p)$$

Of course, it is not at all easy to see that this is equivalent to the original program. But it is clear from this point-wise expression that, for instance, the addition function is an instance: take for $T$ the identity function and define $U.(q,p)$ as $q+1$. So, the program scheme has useful applications.

We can make the following calculation, which gives another way of interpreting program (7.31).

$$\begin{array}{ll}
 & F.(I{\times}S)\times I \circ \mathsf{pass} \vartriangle \gg \circ F.A \times B \\
\subseteq & \qquad \{ \text{ distribution of composition over split } \} \\
 & F.(I{\times}S)\times I \circ (\mathsf{pass} \circ F.A \times B) \vartriangle (\gg \circ F.A \times B) \\
\subseteq & \qquad \{ \text{ assumption (7.32); (2.51) } \} \\
 & F.(I{\times}S)\times I \circ (F.(A{\times}B) \circ \mathsf{pass}) \vartriangle (B \circ \gg) \\
= & \qquad \{ \times\text{-}\vartriangle\text{-fusion } \} \\
 & F.(I{\times}S)\times I \circ F.(A{\times}B)\times B \circ \mathsf{pass} \vartriangle \gg \\
= & \qquad \{ \text{ relators distribute over composition } \} \\
 & F.(A{\times}(S \circ B))\times B \circ \mathsf{pass} \vartriangle \gg \\
\subseteq & \qquad \{ \ A \subseteq I \text{ and domains } \} \\
 & F.(A \times (S \circ B)_{<})\times B \circ F.(I{\times}S)\times I \circ \mathsf{pass} \vartriangle \gg
\end{array}$$

If $A$ is interpreted as $n-1$ and $B$ as the parameter $p$ it might be a lot clearer what is passed to the recursive call: the pair consisting of $n-1$ and the output of $S$ for input $p$. This is in general a good technique to mimic point-wise reasoning in the relation calculus: use monotypes and interpret them as points.

After all this interpreting, the reader may find the proof of termination very easy. First observe that relation $\mathsf{in}^{\cup}$ is $\mathsf{F}$-reductive. Next, note that from the above calculation it follows that

$$\mathsf{F}.(\mathrm{I}{\times}\mathsf{S}) \times \mathrm{I} \circ \mathsf{pass} \vartriangle\!\gg \circ \mathsf{F}.\mathsf{A} \times \mathrm{I} \subseteq \mathsf{F}.(\mathsf{A}{\times}\mathrm{I}) \times \mathrm{I} \circ \mathsf{F}.(\mathrm{I}{\times}\mathsf{S}) \times \mathrm{I} \circ \mathsf{pass} \vartriangle\!\gg \quad .$$

In other words: relation $\mathsf{F}.(\mathrm{I}{\times}\mathsf{S}) \times \mathrm{I} \circ \mathsf{pass} \vartriangle\!\gg$ is a natural transformation on monotypes of type $(\times\mathrm{I}) \circ \mathsf{F} \dot{\leadsto} ((\times\mathrm{I}) \circ \mathsf{F}) \circ (\times\mathrm{I})$. Furthermore, $(\times\mathrm{I})$ is a relator which distributes over all unions of monotypes. By theorem 6.14 it now follows that $\mathsf{F}.(\mathrm{I}{\times}\mathsf{S}) \times \mathrm{I} \circ \mathsf{pass} \vartriangle\!\gg \circ \mathsf{in}^{\cup} \times \mathrm{I}$ is a $((\times\mathrm{I}) \circ \mathsf{F})$-reductive relation. Hence, by definition, program (7.31) is a terminating program.

As a final example of a termination proof we show that the selection sort program terminates. This program is:

$$\mathsf{slsrt} \;=\; \mathsf{nil} \triangledown \mathsf{cons} \circ \mathbb{1}{+}\mathrm{I}{\times}\mathsf{slsrt} \circ \mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{select}) \quad . \tag{7.33}$$

Relation $\mathsf{select}$ holds between two lists if the output list has the property that it can be obtained from the input list by swapping the first element and the minimum of the list. Because there is no plumbing relation involved the program is easy to interpret: it relates the empty list to the empty list. A non-empty list is sorted by swapping the first element and the minimum of the input list ($\mathsf{select}$), then the list is taken apart into the head and the tail ($\mathsf{cons}^{\cup}$), the tail is sorted recursively ($\mathrm{I}{\times}\mathsf{slsrt}$), finally the head, i.e. the minimum of the input, is consed to the result of the recursive call ($\mathsf{cons}$).

It is clear that the program terminates: the argument of the recursive call is a shorter list than the input of the program. Now we might try to rewrite $\mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{select})$ into the form $\mathsf{U} \circ \mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup}$ and use corollary 6.15 to prove termination. However, the elements of the list passed as argument to the recursive call may differ from the elements of the tail of the input list. This means that relation $\mathsf{U}$ will be a relation from and to $(\mathbb{1}{+}\mathrm{I}\times)$-structures over lists such that the contents of the output $(\mathbb{1}{+}\mathrm{I}\times)$-structure (i.e the argument of the recursive call) may differ from the contents of the input structure of $\mathsf{U}$ (i.e. the tail of the input of the program). As a consequence, 6.15 is not applicable, because one of the conditions is that $\mathsf{U}$ must be a natural transformation, i.e. $\mathsf{U}$ should not affect the contents of the structures it acts on.

The termination proof of the selection sort program depends on the observation that $\mathsf{select}$ is a relation between lists of equal length. The largest relation between lists of equal length is $\mathsf{list}.\top\!\!\top$: this relation holds between lists of equal length such that the elements of the input and output list are related by the total relation, which means that the only thing we can say about the input and output is that they are of equal length. In fact, the relation $\mathsf{list}.\top\!\!\top$ can be used to formalise the notion "equal length": relation $\mathsf{R}$ is a relation between lists of equal length iff $\mathsf{R}$ is contained in $\mathsf{list}.\top\!\!\top$. Now we can calculate:

$$\begin{array}{ll} & \mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{select}) \\ \subseteq & \qquad \{ \ \mathsf{select} \text{ is a relation between lists of equal length} \ \} \\ & \mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{list}.\top\!\!\top) \end{array}$$

$$
\begin{array}{ll}
= & \{\ \ \mathsf{list}.R \circ \mathsf{nil} = \mathsf{nil}\text{, i.e. } \mathsf{list}.R \text{ maps the empty list to the empty} \\
& \quad \text{list; this is used in the “converse” form} \\
& \quad \mathsf{nil}^{\cup} \circ (\mathsf{list}.\top\!\!\top)^{\cup} = \mathsf{nil}^{\cup}; \\
& \quad \text{relators commute with converse; } \top\!\!\top^{\cup} = \top\!\!\top \ \ \} \\
& (\mathsf{nil}^{\cup} \circ \mathsf{list}.\top\!\!\top) \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{list}.\top\!\!\top) \\
= & \{\ \ \blacktriangledown\text{-}\circ\text{-fusion}\ \ \} \\
& \mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup} \circ \mathsf{list}.\top\!\!\top
\end{array}
$$

By theorem 6.10 relation $\mathsf{nil}^{\cup} \blacktriangledown (\mathsf{cons}^{\cup} \circ \mathsf{select})$ is $(\mathbb{1}+I\times)$-reductive if $\mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup} \circ \mathsf{list}.\top\!\!\top$ is, that is the sorting program is terminating if we can prove that $\mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup} \circ \mathsf{list}.\top\!\!\top$ is a reductive relation. It might be clear from interpretation that this is true because $\mathsf{nil}^{\cup} \blacktriangledown \mathsf{cons}^{\cup}$ is reductive. However, what is needed to give a rigorous proof is the following general theorem on map relators. (The theorem could not be given in the previous chapter because it mentions map relators.)

The instantiations to obtain the desired result from the theorem are: take $\mathsf{list}$ for $\mathsf{map}$, $\mathbb{1}+(R\times S) = R\oplus S$, and $\mathsf{nil} \triangledown \mathsf{cons} = \psi$.

**Theorem 7.34** Let $\oplus$ be a binary relator, $\psi$ an initial $(I\oplus)$-algebra, and $\mathsf{map}$ the map relator corresponding to $\oplus$ and $\psi$. Then $\psi^{\cup} \circ \mathsf{map}.\top\!\!\top$ is $(I\oplus)$-reductive.

**Proof** We prove reductivity using characterisation (6.5)(c).

$$
\begin{array}{ll}
& (\psi^{\cup} \circ \mathsf{map}.\top\!\!\top) \backslash (I\oplus A) \subseteq A \\
\Rightarrow & \{\ \ \text{monotonicity of } R\backslash\ \ \} \\
& \mathsf{map}.\top\!\!\top \backslash ((\psi^{\cup} \circ \mathsf{map}.\top\!\!\top) \backslash (I\oplus A)) \subseteq \mathsf{map}.\top\!\!\top \backslash A \\
\equiv & \{\ \ \text{monofactors (2.47)}\ \ \} \\
& (\psi^{\cup} \circ \mathsf{map}.\top\!\!\top \circ \mathsf{map}.\top\!\!\top) \backslash (I\oplus A) \subseteq \mathsf{map}.\top\!\!\top \backslash A \\
\equiv & \{\ \ \psi^{\cup} \circ \mathsf{map}.\top\!\!\top \ = \ \top\!\!\top \oplus \mathsf{map}.\top\!\!\top \circ \psi^{\cup} \\
& \quad \text{to check this it is needed to know that } \psi^{\cup} \circ \psi = \mathsf{map}.I\ \ \} \\
& (\top\!\!\top \oplus \mathsf{map}.\top\!\!\top \circ \psi^{\cup}) \backslash (I\oplus A) \subseteq \mathsf{map}.\top\!\!\top \backslash A \\
\equiv & \{\ \ \text{monofactors (2.47)}\ \ \} \\
& \psi^{\cup} \backslash ((\top\!\!\top \oplus \mathsf{map}.\top\!\!\top) \backslash (I\oplus A)) \subseteq \mathsf{map}.\top\!\!\top \backslash A \\
\Rightarrow & \{\ \ \text{relators distribute over monotype factors (2.60);} \\
& \quad \text{this also holds for binary relators}\ \ \} \\
& \psi^{\cup} \backslash ((\top\!\!\top \backslash I) \oplus (\mathsf{map}.\top\!\!\top \backslash A)) \subseteq \mathsf{map}.\top\!\!\top \backslash A \\
\equiv & \{\ \ R\backslash I = I\ \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \psi^{\cup} \searrow (\mathrm{I}\oplus(\mathsf{map}.\top\!\!\top \searrow A)) \subseteq \mathsf{map}.\top\!\!\top \searrow A \\
\Rightarrow & \quad \{ \quad \psi \text{ is an initial algebra, hence } \psi^{\cup} \text{ is reductive} \quad \} \\
 & \mathrm{I} \subseteq \mathsf{map}.\top\!\!\top \searrow A \\
\equiv & \quad \{ \quad \text{monotype factor shunting} \quad \} \\
 & (\mathsf{map}.\top\!\!\top)< \subseteq A \\
\equiv & \quad \{ \quad \text{relators commute with the domain operator;} \\
 & \qquad \top\!\!\top = \top\!\!\top^{\cup}, \text{ so } \top\!\!\top< = \top\!\!\top> \quad \} \\
 & (\mathsf{map}.\top\!\!\top)> \subseteq A \\
\Rightarrow & \quad \{ \quad \text{domains: } (R \circ S)> \subseteq S> \quad \} \\
 & (\psi^{\cup} \circ \mathsf{map}.\top\!\!\top)> \subseteq A
\end{array}
$$

□

In this section we gave a number of termination proofs. The problem with these proofs is not that they are difficult: in fact they are all very short and easy. The problem is that they are a posteriori proofs: they were only verifications. But what we really want is that the theory of reductivity can be used to *construct* terminating programs.

That reductivity arguments can help in the construction of programs is clear from the standard way to construct repetitions. After one has chosen invariant, guard and bound function, the construction of the body is lead by the fact that the bound must decrease: i.e. there must be a reductive part in the body. Once such a reductive part is chosen, the overall shape of the body is known. The design of the remaining parts of the body is then almost dictated by the requirement that the invariant must be maintained.

A similar method is possible for the design of hylo programs. Suppose for instance that the input of the program is of type $\mathsf{map}.\mathrm{I} \times \mathrm{I}$, and that we have decided that the relator is $F$. The program is thus of the form

$$X = S \circ F.X \circ R$$

We must now construct $R$ in such a way that it has a reductive part. Because one of the inputs of the program is of type $\mathsf{map}.\mathrm{I}$, a standard choice would be the converse of an initial $(\mathrm{I}\oplus)$-algebra, i.e. $\psi^{\cup}$. (This standard choice corresponds to the choice "increase a counter" for the repetition.) To obtain a larger reductive part (that is a reductive part that admits more implementations), we could also, on behalf of theorem 7.34, consider the candidate $\psi^{\cup} \circ \mathsf{map}.\top\!\!\top$. It is clear from typing considerations that $R$ has the form $U \circ \psi^{\cup} \times \mathrm{I}$, where $U$ is a relation "to get the types right". From theorem 6.14 we get another restriction on $U$: it has to be a natural transformation. The overall shape of $U$ is then almost dictated by the particular form of $F$ and $\oplus$. Of course one has to make $U$ as general as possible so as to leave some freedom to fill in the unknown parts.

## 7.7 Intermediate data structures

Besides the interpretation of the relator in a hylo equation as a representation of a particular solution strategy, there is another view, which may provide an illuminating insight into programming methodology. The essence is captured by a theorem that states that the least solution of a hylo equation is a composition of two special relations. The theorem is as follows.

**Theorem 7.35** Let $\mathsf{F}$ be a relator with initial algebra $\mathsf{in}$. Then we have for all relations $\mathsf{R}$ and $\mathsf{S}$

$$[\![\mathsf{R},\mathsf{S}]\!] = [\![\mathsf{R}\,,\mathsf{in}^{\cup}]\!] \circ [\![\mathsf{in},\mathsf{S}]\!]$$

**Proof** We prove the stronger $[\![\mathsf{R},\mathsf{S}]\!] = [\![\mathsf{R}\,,\mathsf{in}^{\cup}]\!] \circ \mathsf{Y}$ for any $\mathsf{Y}$ satisfying

$$\mathsf{Y} = \mathsf{in} \circ \mathsf{F.Y} \circ \mathsf{S} \quad , \tag{7.36}$$

i.e. the requirement that $\mathsf{Y}$ is a least solution is dropped. Because initial algebras can be "shifted" (the converse dual of (6.31)) equation (7.36) implies

$$\mathsf{in} \circ \mathsf{Y} = \mathsf{F.Y} \circ \mathsf{S} \quad . \tag{7.37}$$

The demonstrandum has a shape which invites the application of $\mu$-fusion.

$$
\begin{array}{ll}
 & [\![\mathsf{R}\,,\mathsf{in}^{\cup}]\!] \circ \mathsf{Y} = [\![\mathsf{R},\mathsf{S}]\!] \\
\Leftarrow & \quad \{ \ \mu\text{-fusion: } (\circ\mathsf{Y}) \text{ has an upper adjoint;} \\
 & \quad\quad \text{for all } \mathsf{X}. \ \} \\
 & \mathsf{R} \circ \mathsf{F.X} \circ \mathsf{in} \circ \mathsf{Y} = \mathsf{R} \circ \mathsf{F.}(\mathsf{X} \circ \mathsf{Y}) \circ \mathsf{S} \\
\Leftarrow & \quad \{ \ \text{relators distribute over composition;} \\
 & \quad\quad \text{monotonicity} \ \} \\
 & \mathsf{in} \circ \mathsf{Y} = \mathsf{F.Y} \circ \mathsf{S} \\
\equiv & \quad \{ \ \text{assumption (7.37)} \ \} \\
 & \text{true}
\end{array}
$$

□

The so-called *anamorphism* $[\![\mathsf{in},\mathsf{S}]\!]$ constructs an element of the carrier set of $\mathsf{in}$, from the input value; the so-called *catamorphism* $[\![\mathsf{R}\,,\mathsf{in}^{\cup}]\!]$ "destructs" that element into an output value of type $\mathsf{B}$. The composition $[\![\mathsf{R}\,,\mathsf{in}^{\cup}]\!] \circ [\![\mathsf{in},\mathsf{S}]\!]$ makes this construction/destruction process explicit. The form of the hylo equation hides this process. In other words, it remains invisible and of no concern to the programmer.

For this reason, the relator $\mathsf{F}$ is said to specify an intermediate (or *virtual* [81]) data structure.

Two simple examples of virtual datatypes are provided by do-statements and the factorial function. In the case of do-statements (see (7.21)) the virtual datatype is the carrier set of an initial (I+)-algebra, a type which is isomorphic to $\mathrm{I}\times\mathsf{Nat}$ —thus an element of the virtual datatype can be seen as a pair consisting of an element of the state space and a natural number, the latter being a "virtual" count of the number of times the loop body is executed. In the case of the factorial function, the standard recursive definition can be rewritten in the form of the hylo equation:

$$\mathsf{fact} \;=\; \mathsf{one}\triangledown(\mathsf{times}\circ\mathsf{succ}\times\mathrm{I})\circ\mathbb{1}+(\mathrm{I}\times\mathsf{fact})\circ\mathsf{zero}^{\cup}\blacktriangledown(\mathrm{I}\vartriangle\mathrm{I}\circ\mathsf{succ}^{\cup}) \quad .$$

The "virtual" datatype is thus the type of lists (the carrier set of an initial $\mathbb{1}+(\mathrm{I}\times)$-algebra). The list that is constructed for a given input $n$ is the list of natural numbers from $n-1$ down to $0$ and the hylo theorem states that the factorial of $n$ can be calculated by constructing this list (the anamorphism phase) and then multiplying the numbers together after adding $1$ (the catamorphism phase).

The design of hylomorphisms combines the techniques from sequential programming with the recursive approach of functional programming. The design of the anamorphism part involves the construction of a "type invariant" and a reductive part whilst the construction of the catamorphism is a recursive problem decomposition, making use of the type invariant.

For example, merge sort consists of two phases. In the first phase a tree structure (with the values in the leaves) is formed with the invariant property that first, the subtrees are balanced and, second, the elements of the subtrees are the elements of the input list. In the second phase the tree is broken down into a sorted list whereby two lists are joined by the standard "merging" process. Of course, in the usual implementations of merge sort the construction of the tree is "virtual", i.e. not made explicit. (However, the tree is implicitly present in the stack used to implement the recursion.)

Quicksort is also an example: the first phase constructs a binary search tree. That is, a binary tree is constructed with the invariant property that the value in the root is at least as big as the maximum value in the left subtree and smaller than the values in the right subtree. Another invariant is that the values in the tree are the values in the input list. It is then easy to construct the program for the second phase that flattens this tree into a sorted list.

For an interesting example in which the intermediate data structure is a heap we refer the reader to [81].

Of course the use of intermediate structures is not restricted to simple hylo programs: the two phases can themselves be arbitrary hylo programs, rather than a cata and an ana. Examples are abundant: for instance heapsort (where the intermediate structure is a heap) and compilers (where it is of course the parse tree). However we have stressed that intermediate structures also play a role in programs like quicksort because most programmers are not aware of the fact.

The examples given above suggest that the programmer should be well aware that a hylo program can be seen as consisting of two phases and that an intermediate datatype is involved. As the quicksort example shows, the two phases of well-known programs can be interesting in their own right: in particular the program that constructs a binary search tree is encountered often in practice. It could also very well be that phases of existing programs can be reused in the design of other programs. Finally, it can be a good method to solve a programming problem by designing the intermediate data structure first, doing this in such a way that it becomes easy to construct the two phases of a hylo program.

## 7.8 A programming paradigm

Programming can be made less error-prone by making it into a more calculational activity. Proving termination being so essential in the design, our ultimate goal is the design of a calculus of terminating programs. This calculus should include a calculus of reductive relations, by means of which proving termination and constructing terminating programs can become a more syntactic activity.

In addition, one might think of program transformations that preserve or even introduce termination. The latter especially might constitute a strong tool, in combination with other techniques like program inversion: designing, for instance, a program that flattens a parse tree, is simpler than designing a parser. The reason is that flattening a parse tree is a process that can be described with structural recursion: the flattener is a cata equation. The converse of a flattening program is a parser. However, this parser is no longer a cata program, so it may very well be that, by inversion of the flattener, termination is lost. Transformations that reduce nondeterminism might then restore termination. (For an example of how a parser can be derived by inverting a flattener see [50]. The presentation there is however not relational.)

In general, the design of a program starts with a specification (relation) $\mathsf{S}$, often expressed using predicates. Relational, i.e. point-free, formulae can be far more compact than predicates, so $\mathsf{S}$ is rewritten into relational form. Another reason to do this rewrite is because it can help to structure the specification. In this process, however, one should not be holier than the pope by insisting that all points be removed. One can, for instance, avoid making the plumbing of rearrangements of arguments and other natural transformations explicit by introducing a name for the plumbing relation and just defining it pointwise; (using the name $\mathsf{swap}$, say, in the relational form of $\mathsf{S}$ and just defining it pointwise as the swap of the elements in a pair is much to be preferred to explicitly having $\gg \circ \ll^{\cup} \cap \ll \circ \gg^{\cup}$ in the rewrite of $\mathsf{S}$).

Once specification $\mathsf{S}$ has been moulded into relational form, program design starts. We have to determine a relator $\mathsf{F}$ and relations $\mathsf{R}$ and $\mathsf{T}$ such that:

$$\mathsf{T} \circ \mathsf{F.S} \circ \mathsf{R} \subseteq \mathsf{S} \qquad \text{, where } \mathsf{R} \text{ is } \mathsf{F}\text{-reductive.} \tag{7.38}$$

The choice of $\mathsf{F}$ corresponds to the choice of a solution strategy like structural recursion, repetition, divide-and-conquer, etc. If it is not clear which solution strategy should be taken, relator $\mathsf{F}$ may be found by choosing a suitable intermediate data structure.

That relation $\mathsf{R}$ has to be reductive should be of great help in its design: somewhere in $\mathsf{R}$ there has to be a reductive part. The typing and properties of the other parts are then almost dictated. (In the standard construction of repetitions this is a well-known phenomenon.) If $\mathsf{R}$ and $\mathsf{T}$ are not implementable, they serve as new specifications to be dealt with separately.

This is what constitutes the core of programming. ( Sometimes additional coding can be necessary to overcome the limitations of a target programming language. Such obligations might be considered an indication of what the features of a programming language should be so as to make it a more convenient language.)

The program designed by solving (7.38) is the terminating program $\mathsf{X} = \mathsf{T} \circ \mathsf{F.X} \circ \mathsf{R}$. If it is executed by a computer, the input-output relation constructed is a —and, hence, *the*— solution of the equation. That this relation does, indeed, satisfy the specification, i.e. is contained in $\mathsf{S}$, is an immediate consequence of Knaster-Tarski and (7.38), since a unique solution is also a least solution.

## 7.9 Concluding remarks

In this chapter we viewed (simple) programs as equations in relations. This is a rather natural view of programs. If we look at a functional program written in Haskell to compute the factorial function it is clear that the program has the form of an equation. Equally clear is that a recursive program written in Pascal can be seen as an equation: the "heading" equals the "body". Programs are equations because that gives the possibility to execute them. The input-output relation that is computed this way is a (the least) solution of the equation. We have demonstrated that if the execution terminates for all inputs there is only one solution of the equation. To state it differently: the relation computed by the program is defined by the equation.

It is important that one can distinguish a program from the relation it computes: in general there are many programs possible to compute a given relation. Modelling programs as equations in relations makes this possible: one can talk about different programs which compute the same function. For instance, there are several equations which have the sorting function as the solution, i.e. there are several sorting programs.

The view of programs as equations has an additional advantage. A specification can be seen as a description of the input-output relation the program has to compute. Therefore, programming is the construction of an equation such that the specification is a solution of that equation. Because a specification is a relation and a program an equation in relations, it is possible (at least in principle) to construct the program in a purely calculational fashion, since the relation calculus can be used.

An issue that was not discussed here is the problem of totality: can a program be started in any state? In the case of a parsing program constructed by inverting a flattener, for instance, not every input can be accepted because not every input string has a parse tree. Capturing this is an additional difficulty not studied in this thesis. However, if the relations $\mathsf{R}$ and $\mathsf{S}$ in hylo program (7.18) have the property that they are total, i.e. all inputs can be accepted, it can be shown that the hylomorphism is total too. In that case, the program as a whole can therefore be started in any state. This explains the remark after theorem 6.21 that totality is a desirable property.

In this chapter, we have confined our attention to simple programs, i.e. hylo programs where the substatements were considered to be basic statements and not defined by recursion. The study of programs that are formed by combining such simple programs, and where the substatements can also be recursively defined is discussed in the next chapter.

# Chapter 8

# The (R,A)-calculus

## 8.1 Introduction

This chapter studies how to determine the input-output relation and termination properties of composite programs. In the previous chapter a recursive program was identified with an equation in relations. However, there were a number of restrictions. First, the program was a single equation: we did not define how to compose programs to obtain larger programs. Second, the "statements" of the body of a recursive program were restricted to simple relations only: it was not discussed what the consequences are of allowing them to be recursively defined as well. These restrictions can be relaxed without any problem as long as one considers terminating programs only. Besides their shape, the only relevant attribute of such programs is the input-output relation. And there can be no confusion about the input-output relation of, for instance, the following composite program.

$$\begin{aligned}
\mathsf{main} &= \mathsf{S} \circ \mathsf{R} \\
\mathsf{S} &= \mathsf{V} \circ \mathsf{F.S} \circ \mathsf{U} \\
\mathsf{U} &= W \circ \mathsf{G.U} \circ \mathsf{T}
\end{aligned}$$

In this program $\mathsf{main}$ is the main procedure consisting of the composition of a simple command $\mathsf{R}$ and a recursively defined procedure $\mathsf{S}$. Procedure $\mathsf{S}$ is defined in terms of recursive procedure $\mathsf{U}$ and simple command $\mathsf{V}$. Finally $\mathsf{U}$ is defined in terms of simple programs. (Note that all three components of the program are hylo programs.) Now suppose that $\mathsf{T}$ is $\mathsf{G}$-reductive. Then relation $\mathsf{U}$ is determined by the third component: $\mathsf{U} = [\![\mathsf{G}; W,\mathsf{T}]\!]$. Likewise, if $\mathsf{U}$ is $\mathsf{F}$-reductive, we have $\mathsf{S} = [\![\mathsf{F}; \mathsf{V},\mathsf{U}]\!]$. So, the input-output relation of the main component is $[\![\mathsf{F}; \mathsf{V},[\![\mathsf{G}; W,\mathsf{T}]\!]]\!] \circ \mathsf{R}$

However, the situation changes if one allows programs that are not necessarily terminating. Then it has to be defined what the safe set of the composite program is. In this chapter it is indicated how this can be done. The material of the chapter has been published before [40] and, for this reason, the terminology and approach differ somewhat from that of the previous chapter.

In this chapter we take the view that a program is a set of computations, and a computation a pair of states. That is, we consider a program to be an input-output relation, rather than an equation in relations. A remaining problem is now what to do with the infinite or diverging computations. One possibility is to forget about them as in [34, 47]. This is known as the *angelic* approach: if a computation may terminate, it will. Another possibility is the *demonic* approach, see for instance [75]: if a computation can diverge it will. We will pursue neither of these, but instead take the so-called *erratic* approach as in [16, 49]: it is possible that a program is both able to terminate and to diverge on a given input. There are at least two existing ways to model this: add a bottom element to the universe and model a program as a so-called extended relation as in [84, 49, 38], or add a set of safe states as is done in [16, 61] and model a program as a relation and a set. These two approaches are really the same: an extended relation can always be transformed into the corresponding relation-set pair, and vice versa. For technical reasons, we prefer the second approach.

To see the link between this approach and the one of chapter 7 one has to observe that the term "program" in that chapter is used to denote the concepts of "recursive procedure" and "recursive function". In programming languages such procedures and functions are defined by an equation

$$\mathfrak{p} = \mathcal{F}.\mathfrak{p}$$

where $\mathfrak{p}$ is the procedure and $\mathcal{F}.\mathfrak{p}$ the body. This makes clear that a recursive procedure is indeed an equation in programs. In short: the notions "program" and "relation" of chapter 7 correspond to the notions "equation in programs" and "program" of this chapter, respectively.

Another difference between the two chapters is the class of equations that is discussed. Chapter 7 studied hylo equations, this chapter equations of the form $\mathfrak{p} = \mathcal{F}.\mathfrak{p}$ where the program transformer $\mathcal{F}$ is built from the usual operators like sequential composition and non-deterministic choice. However, in $\mathcal{F}$ there are no operators involved that correspond to relators. We have not introduced relator-like operators on the programs of this chapter because there are still some difficulties to be resolved. Specifically, the obvious generalisation of relator (maybe it should be called a programmator) does not distribute unconditionally over sequential composition.

However, although there is the restriction that the equations of this chapter do not involve relators, there is also an extension with respect to the equations of the previous chapter. In chapter 2 it was mentioned that when relations are seen as programs, relation union corresponds to the nondeterministic choice operator. The approach of this chapter has the advantage that it allows the introduction of another choice operator that can not be defined when programs are seen as just relations. We call this operator the fair choice operator. To our knowledge this operator is absent from existing (imperative) programming languages and calculi.

An explanation for its absence might be that the operator is not monotone with respect to the so-called Egli-Milner order. Moreover, this order is the standard vehicle in the treatment of recursion, see [73, 16]. So, we had the choice of either dropping the operator and sticking to the standard approach to recursion, or adopting the operator and trying to find a novel approach to recursion. Fortunately, the latter choice turned out to be possible.

In conclusion, this chapter contains a treatment of recursion that seems to be new. Basically, we deviate from the standard approach in that we do not define a recursive program (i.e. a relation-set pair) as a least fixed point. We construct the constituent parts of a program more or less independently instead. First, the relation part of the program is constructed as a least fixed point, and then the set part of the program is defined, also as a least fixed point. Usually, the function of which the set part is the least fixed point has the relation part of the program to be constructed as a parameter.

Of course, when one introduces such a novel method, the question arises whether it is compatible with the standard approach via the Egli-Milner order. The chapter contains the (positive) answer to that question as well.

## 8.2 Programs and monotype transformers

As we have mentioned in the introduction, we model a program by an input-output relation, together with a set $S$ of states such that for any $x \in S$ the program will not "diverge" when started in $x$. Because we will need to combine the set and the relation (for instance by restricting the domain of the relation to the set) it is convenient to code the set as a relation. There are at least two equivalent ways to do so: we can take a so-called vector (i.e. a relation $R$ with $R = R \circ \top\!\top$) as in [61, 16] or a monotype as in [12]. For technical reasons we prefer the latter: we identify a set with its identity relation.

**Definition 8.1 (Program)** Let $\mathcal{R}$ be a relation algebra. A program over $\mathcal{R}$ is a pair $(R, A)$, where $R$ is a relation from $\mathcal{R}$, and $A$ a monotype from $\mathcal{R}$.
□

In [24], relation $R$ is called the breadth and $A$ the definedness of the program $(R, A)$. We call the relation and the monotype the i-o relation and the safe monotype of the program, respectively. In previous chapters we had the convention that the input of relations is from the right, because that corresponds to the convention of functional programming languages. In this chapter we give a semantics of an imperative language and the standard convention for imperative programs is to have input on the left side. Therefore, we adopt in this chapter the convention that relations have input on the left side and output on the right side.

To give the link between the view of programs as a pair and predicate transformer semantics, we now define the functions *wp* and *wlp*. The function wlp.$(S, A)$ is a function from monotypes to monotypes (a monotype transformer) that maps a monotype $B$ to the greatest monotype $C$ such that each element of $C$ is related by $S$ to elements of $B$ only, or much more compactly expressed:

$$C \subseteq \mathrm{wlp}.(S, A).B \quad \equiv \quad (C \circ S)_{>} \subseteq B \quad . \tag{8.2}$$

From this we see immediately that wlp.$(S, A)$ is the upper adjoint of the function $_{>} \circ (\circ S)$, i.e. the function $(\not{/} S)$. So we define:

**Definition 8.3 (weakest liberal precondition)** For program $(\mathsf{S},\mathsf{A})$ and monotype $\mathrm{B}$:

$$\mathrm{wlp}.(\mathsf{S},\mathsf{A}).\mathrm{B} = \mathrm{B}\not/\mathsf{S}$$

□

Next, we turn to the definition of the weakest precondition. Of this monotype transformer only its value in $\mathrm{I}$ is interesting since we define:

**Definition 8.4 (termination law)**

$$\mathrm{wp}.(\mathsf{S},\mathsf{A}).\mathrm{B} \quad = \quad \mathrm{wlp}.(\mathsf{S},\mathsf{A}).\mathrm{B} \;\cap\; \mathrm{wp}.(\mathsf{S},\mathsf{A}).\mathrm{I}$$

□

(Note that this is possible because $\mathrm{wlp}.(\mathsf{S},\mathsf{A}).\mathrm{I} = \mathrm{I}\not/\mathsf{S} = \mathrm{I}$.) This leaves us with the task of defining the monotype $\mathrm{wp}.(\mathsf{S},\mathsf{A}).\mathrm{I}$ . This monotype is the greatest monotype $\mathsf{B}$ such that $(\mathsf{S},\mathsf{A})$ will not diverge when started in a state from $\mathrm{B}$, i.e. the safe set $\mathsf{A}$.

**Definition 8.5 (weakest precondition)**

$$\mathrm{wp}.(\mathsf{S},\mathsf{A}).\mathrm{I} = \mathsf{A}$$

□

Combining the definitions 8.4, 8.3 and 8.5 we get the following result:

$$\mathrm{wp}.(\mathsf{S},\mathsf{A}).\mathrm{B} \quad = \quad \mathrm{B}\not/\mathsf{S} \;\cap\; \mathsf{A} \quad . \tag{8.6}$$

Now we can check the healthiness conditions as demanded by [35]. The first, stating that the function $\mathrm{wlp}.(\mathsf{S},\mathsf{A})$ distributes over arbitrary intersections of monotypes (is universally conjunctive in the terminology of [37]), is immediate because it is an upper adjoint. The second condition, known as the termination law, is true by definition as it is just definition 8.4. Of course, it is not strange that these conditions hold, because it is well known that they are imposed in predicate transformer semantics to ensure that there exists an equivalent relational semantics.

However, the third healthiness condition, known as the law of the excluded miracle, is not true in general. This law states that $\mathrm{wp}.(\mathsf{S},\mathsf{A}).\perp\!\!\!\perp = \perp\!\!\!\perp$.

$$\begin{array}{ll}
 & \mathrm{wp}.(\mathsf{S},\mathsf{A}).\perp\!\!\!\perp = \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; (8.6);\ \perp\!\!\!\perp \text{ is bottom element } \;\; \} \\
 & \perp\!\!\!\perp\not/\mathsf{S} \cap \mathsf{A} \subseteq \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; \text{the dual of } (2.40) \;\; \} \\
 & \sim(\mathsf{S}{<}) \cap \mathsf{A} \subseteq \perp\!\!\!\perp \\
\equiv & \quad \{ \;\; \text{shunting} \;\; \} \\
 & \mathsf{A} \subseteq \mathsf{S}{<}
\end{array}$$

Programs that do not satisfy the law of the excluded miracle were first introduced in [73]. Following [49] we call programs that do satisfy the law "total".

**Definition 8.7 (Totality)** A program $(S, A)$ is total iff it satisfies $A \subseteq S<$.
□

That we allow non-total programs also explains why we say "will not diverge" instead of "are guaranteed to terminate" in the introduction of the notion safe monotype. We have a number of cases:

- Start program $(S, A)$ in $\sim A \cap \sim(S<)$ : divergence is certain.
- Start program $(S, A)$ in $\sim A \cap S<$ : divergence and termination are both possible.
- Start program $(S, A)$ in $A \cap S<$ : termination is certain.
- Start program $(S, A)$ in $A \cap \sim(S<)$ : divergence is impossible, but so is termination: the start "fails".

Non-total programs are programs for which there are states from which it is impossible to start the program. Therefore, they are not implementable, they are not "code" as [71] expresses it. However, as argued in [73] they are useful in a calculus of programs.

From the theory of predicate transformer semantics [49] we expect that there is a 1-1 correspondence between the monotype transformers that distribute over arbitrary intersections and the relations. It is indeed possible to prove this using point-wise reasoning. It can also be proved using relational calculus. However, to do so we need the axiom of extensionality. Roughly, this axiom states that a relation is determined by its "action on monotypes", and it has many equivalent formulations (for a discussion of extensionality and a number of equivalent formulations see [76]). The formulation that is needed to prove the aforementioned 1-1 correspondence is as follows.

**Definition 8.8 (Extensionality)** A relation algebra is extensional iff for all relations $R$ and $S$:

$$R = S \equiv \forall(A\colon A \subseteq I\colon A/R = A/S)$$

□

## 8.3 The straightline constructs

In this section we define some programs and operators on programs but without recursive constructs. These are the programs that [37] calls "straightline".

### 8.3.1 Extremal programs

Recall that a program is a pair $(\mathsf{R}\,,\mathsf{A})$, where $\mathsf{R}$ is a relation and $\mathsf{A}$ a monotype. As the first examples of programs we consider those that are obtained by taking for $\mathsf{R}$ and $\mathsf{S}$ the appropriate constants of the relation algebra. This gives us as choice for $\mathsf{R}$: $\perp\!\!\perp$, I and $\top\!\!\top$. For $\mathsf{A}$ the monotypes $\perp\!\!\perp$ and I are possible. So, now we have six programs.

- $(\perp\!\!\perp\,,\mathrm{I})$: this program has an empty i-o relation: it can't produce output. Furthermore, its safe monotype is I hence it can't diverge either: it can't do anything. We call this program $\mathsf{miracle}$ (following [72]). In the literature it is also known under the names "fail" [73] and "magic" [4]. To give the link with predicate transformer semantics: we have $\mathrm{wp}.\mathsf{miracle}.\mathsf{A} = \mathrm{wlp}.\mathsf{miracle}.\mathsf{A} = \mathrm{I}$. Note that $\mathsf{miracle}$ is not total: it is indeed a miracle.

- $(\perp\!\!\perp\,,\perp\!\!\perp)$: the interpretation of this one is clear: its i-o relation is empty so it can't produce output, and it can diverge everywhere, because its safe monotype is $\perp\!\!\perp$. Hence it is guaranteed to diverge everywhere. It is known under the name $\mathsf{abort}$ [37] and so it will be called here also. Another name for it is "loop" [73] and this is probably better: the name abort wrongly suggests an immediate termination (for instance because an exception has occurred). It satisfies $\mathrm{wp}.\mathsf{abort}.\mathsf{A} = \mathrm{wlp}.\mathsf{abort}.\mathsf{A} = \perp\!\!\perp$

- $(\top\!\!\top\,,\mathrm{I})$: a program that "can do anything" (set all its variables to arbitrary values) except diverge. It is what [37] calls $\mathsf{havoc}$, and we have $\mathrm{wp}.\mathsf{havoc}.\mathsf{A} = \mathrm{wlp}.\mathsf{havoc}.\mathsf{A}$ and $\mathrm{wp}.\mathsf{havoc}.\mathrm{I} = \mathrm{I}$, whereas $\mathrm{wp}.\mathsf{havoc}.\mathsf{A} = \perp\!\!\perp$ for $\mathsf{A}$ different from I.

- $(\top\!\!\top\,,\perp\!\!\perp)$: yet worse than $\mathsf{havoc}$. It also has the possibility to diverge. Therefore it is appropriate to call it $\mathsf{chaos}$. Program $\mathsf{chaos}$ satisfies: $\mathrm{wlp}.\mathsf{chaos}.\mathsf{A} = \mathrm{wlp}.\mathsf{havoc}.\mathsf{A}$ and $\mathrm{wp}.\mathsf{chaos}.\mathsf{A} = \perp\!\!\perp$.

- $(\mathrm{I}\,,\mathrm{I})$: this is the well known program $\mathsf{skip}$, it can't diverge and does not change the state. For $\mathsf{skip}$ it is the case that $\mathrm{wp}.\mathsf{skip}.\mathsf{A} = \mathrm{wlp}.\mathsf{skip}.\mathsf{A} = \mathsf{A}$.

- $(\mathrm{I}\,,\perp\!\!\perp)$: For this one we have not been able to find a name in the literature. It is a rather uninteresting program: operationally, it is a nondeterministic choice between $\mathsf{skip}$ and $\mathsf{abort}$.

The properties of the *wlp*- and *wp*-functions applied to these programs are all easily checked using definition 8.3, property (8.6), and the properties of the monotype factor.

### 8.3.2 Guards and assertions

The programs $\mathsf{miracle}$ and $\mathsf{skip}$ are special cases of the so-called guards: a guard is a program with as i-o relation a monotype and as safe monotype I. These programs are called guards because they can be used to "implement" Dijkstra's guarded commands $\mathsf{b} \to \mathsf{s}$, where $\mathsf{b}$ is a predicate and $\mathsf{s}$ a statement. Like [49] we write this as $?\mathsf{b};\mathsf{s}$, with ; the sequential composition to be introduced shortly, $\mathsf{s}$ is a program and $?\mathsf{b} = (\mathsf{B}\,,\mathrm{I})$ for $\mathsf{B}$ the monotype corresponding to the predicate $\mathsf{b}$. Because the predicates false and true correspond to the monotypes $\perp\!\!\!\perp$ and I respectively, we have that ?false = **abort** and ?true = **skip**. In general guards are not total, in fact the only total guard is **skip**.

As a sort of dual to guards we have assertions. In [49] the syntax !b is used for predicate $\mathsf{b}$, and the interpretation is: evaluate $\mathsf{b}$, if this yields false then **abort** else **skip**. Therefore, we should take $!\mathsf{b} = (\mathsf{B}\,,\mathsf{B})$ where $\mathsf{B}$ is the monotype corresponding to predicate $\mathsf{b}$. So an assertion is a program with equal i-o relation and safe monotype. Note that !false = **abort** and !true = **skip**. Also note that **skip** is the only program that is both a guard and an assertion.

### 8.3.3 Sequential composition

The operational interpretation of the sequential composition $(\mathsf{R}\,,\mathsf{A});(\mathsf{S}\,,\mathsf{B})$ is of course: first execute $(\mathsf{R}\,,\mathsf{A})$ and then $(\mathsf{S}\,,\mathsf{B})$. We could now proceed by constructing from this interpretation a definition for sequential composition, but it is nicer to derive such a definition from the "specification"

$$\mathrm{wlp}.((\mathsf{R}\,,\mathsf{A});(\mathsf{S}\,,\mathsf{B})).\mathsf{C} = \mathrm{wlp}.(\mathsf{R}\,,\mathsf{A}).(\mathrm{wlp}.(\mathsf{S}\,,\mathsf{B}).\mathsf{C}) \quad \text{and,} \tag{8.9}$$

$$\mathrm{wp}.((\mathsf{R}\,,\mathsf{A});(\mathsf{S}\,,\mathsf{B})).\mathrm{I} = \mathrm{wp}.(\mathsf{R}\,,\mathsf{A}).(\mathrm{wp}.(\mathsf{S}\,,\mathsf{B}).\mathrm{I}) \quad . \tag{8.10}$$

This we do in the following calculations. First we concentrate on requirement (8.9).

$$\begin{aligned}
& \mathrm{wlp}.(\mathsf{R}\,,\mathsf{A}).(\mathrm{wlp}.(\mathsf{S}\,,\mathsf{B}).\mathsf{C}) \\
= & \qquad \{ \ \text{definition 8.3} \ \} \\
& \mathrm{wlp}.(\mathsf{R}\,,\mathsf{A}).(\mathsf{C}/\mathsf{S}) \\
= & \qquad \{ \ \text{definition 8.3} \ \} \\
& (\mathsf{C}/\mathsf{S})/\mathsf{R} \\
= & \qquad \{ \ (2.47) \ \} \\
& \mathsf{C}/(\mathsf{R}\circ\mathsf{S}) \\
= & \qquad \{ \ \text{definition 8.3} \ \} \\
& \mathrm{wlp}.(\mathsf{R}\circ\mathsf{S}\,,?).\mathsf{C}
\end{aligned}$$

In the last line of the above calculation the question mark stands for a "don't care". This calculation only gives a restriction on the i-o relation of the program. Let us see whether the remaining requirement (8.10) yields a solution for the safe monotype.

$$\begin{aligned}
& \mathrm{wp}.(\mathsf{R}\,,\mathsf{A}).(\mathrm{wp}.(\mathsf{S}\,,\mathsf{B}).\mathrm{I}) \\
= & \qquad \{ \ \text{definition 8.5} \ \} \\
& \mathrm{wp}.(\mathsf{R}\,,\mathsf{A}).\mathsf{B}
\end{aligned}$$

$$
\begin{array}{ll}
= & \{ \ (8.6) \ \} \\
 & \mathrm{wlp}.(R\,,A).B \cap \mathrm{wp}.(R\,,A).I \\
= & \{ \ \text{definition } (8.3) \ \} \\
 & B \mathbin{\underline{/}} R \cap A \\
= & \{ \ (8.6) \ \} \\
 & \mathrm{wp}.(? \,,\, B \mathbin{\underline{/}} R \cap A).I
\end{array}
$$

So, combining the two results, we see that there is only one definition for the operator ";" possible such that it satisfies (8.9) and (8.10).

**Definition 8.11 (Sequential composition)** The sequential composition operator ";" on programs is defined as:

$$(R\,,A);(S\,,B) = (R \circ S \ ,\ B \mathbin{\underline{/}} R \cap A)$$

□

Of course, we must now check that the operator ";" has the other desired properties, for instance that it is associative and has skip as a left and right unit. This can be done by straightforward calculations in the relational calculus.

### 8.3.4 The choice operators

Recall that a program is a relation-monotype pair. Because both the relations and the monotypes form a (complete) lattice this suggests four operators on bags of programs.

- Take the intersection in both coordinates.
- Take the intersection in the first and the union in the second coordinate.
- Take the union in the first and the intersection in the second coordinate.
- Take the union in both coordinates.

Of these the first two are traditionally regarded as unattractive. This is due to the intersection in the first coordinate: such operators have the disadvantage that they may not be implementable even if their operands are.

We will, accordingly, not investigate these constructs any further. However, we would like to remark at this point that admitting these operators will not cause any problems in connection with recursion.

The third operator is more familiar: it is the nondeterministic choice.

**Definition 8.12 (Nondeterministic choice)** Let $(R_i\,,A_i)$ be a bag of programs. Then

$$[\!]\,(i :: \ (R_i\,,A_i)) \ \triangleq \ (\cup(i :: \ R_i)\,,\, \cap\,(i :: \ A_i))$$

□

As is usual, we write the operator $[]$ for finite bags as an infix operator:

$$(R,A) [] (S,B) = (R \cup S, A \cap B) \quad . \tag{8.13}$$

The operational interpretation of the operator is: choose one of the components and execute it. We have:

$$\mathrm{wlp}.((R,A) [] (S,B)).C \quad = \quad \mathrm{wlp}.(R,A).C \cap \mathrm{wlp}.(S,B).C \quad , \tag{8.14}$$

and the same for the function *wp*. So it corresponds to the operator $[]$ from [49].

The final operator we introduce is the most interesting because it is not a familiar operator on programs, we call it the *fair choice* operator.

**Definition 8.15 (Fair choice)** Let $(R_i, A_i)$ be a bag of programs then

$$\diamond(i :: (R_i, A_i)) \;\triangleq\; (\cup(i :: R_i), \cup(i :: A_i))$$

□

Again, for finite bags $\diamond$ is used as an infix operator:

$$(R,A) \diamond (S,B) = (R \cup S, A \cup B) \quad . \tag{8.16}$$

The operational interpretation is: make a copy of the initial state, start all components in that state (so there is no interference between the components). Whenever one of the components terminates, its final state is also the final state of the fair choice. Should more than one component terminate, the final state is chosen nondeterministically from their final states.

As far as we are aware there are no programming languages that feature this construct, maybe because of its inherent inefficiency. Nevertheless it is useful in specifications. For instance, suppose we have a function $f$ of type integer $\to$ boolean and we know that for some $n$ it is the case that $f.n = \text{true}$. Then the following program always terminates and has as postcondition $f.a = \text{true}$.

$$\mathsf{do}\ \neg f.a \to a := a{+}1\ \mathsf{od} \quad \diamond \quad \mathsf{do}\ \neg f.a \to a := a{-}1\ \mathsf{od}$$

(We have not defined the $\mathsf{do}$ – $\mathsf{od}$ operator, but it should be interpreted in the standard way.)
The operator $\diamond$ has properties that differ from the other operators considered thus far.

$$\mathrm{wlp}.((R,A) \diamond (S,B)).C \quad = \quad \mathrm{wlp}.(R,A).C \cap \mathrm{wlp}.(S,B).C \quad \text{and}$$

$$\begin{aligned} \mathrm{wp}.((R,A) \diamond (S,B)).C \quad = \quad & (\mathrm{wp}.(R,A).I \cup \mathrm{wp}.(S,B).I) \\ & \cap\, \mathrm{wlp}.(R,A).C \cap \mathrm{wlp}.(S,B).C \end{aligned}$$

But, in contrast to the other operators, the function $\mathrm{wp}.((\mathsf{R},\mathsf{A})\diamond(\mathsf{S},\mathsf{B}))$ cannot be expressed in terms of $\mathrm{wp}.(\mathsf{R},\mathsf{A})$ and $\mathrm{wp}.(\mathsf{S},\mathsf{B})$. In other words: the function *wp* does not distribute over the fair choice operator. To state it yet differently: if the fair choice is admitted as an operator, the function *wp* is not a homomorphism (as in [49]) anymore.

The fair choice operator seems to be a relatively novel element in program calculi. It differs from the operator denoted by the same symbol in [48] because *wp* does distribute over that one: $\mathrm{wp}.(\mathsf{r}\diamond\mathsf{s}).\mathsf{p} \;=\; \mathrm{wp}.\mathsf{r}.\mathsf{p}\vee\mathrm{wp}.\mathsf{s}.\mathsf{p}$. Furthermore, that operator does not have an operational interpretation. Also, the fair choice is not the same as Nelson's operator [73]: first of all that operator is not symmetric and, secondly it has different monotonicity properties. However, recently Broy and Nelson [25] have proposed a similar operator. Hesselink [51] has shown how the semantics of the fair choice operator can be defined with predicate transformers.

## 8.4 Partial orders on programs

Partial orders on programs can be divided into two classes: refinement orders and approximation orders. The idea behind the first is that for such an order $\sqsubseteq$ the expression $\mathsf{r}\sqsubseteq\mathsf{s}$ is interpreted as: "program $\mathsf{r}$ is an implementation of program $\mathsf{s}$", or as "$\mathsf{r}$ is better than $\mathsf{s}$." In general a program with i-o relation $\mathsf{R}$ is considered to be better than a program with i-o relation $\mathsf{S}$ whenever $\mathsf{R}\sqsubseteq\mathsf{S}$. That is when it is less nondeterministic. Also it is considered to be better to have a larger safe monotype. So, there are two orders that come to mind.

$$(\mathsf{R},\mathsf{A})\sqsubseteq_0(\mathsf{S},\mathsf{B}) \;\equiv\; \mathsf{R}\subseteq\mathsf{S} \quad \text{and} \tag{8.17}$$

$$(\mathsf{R},\mathsf{A})\sqsubseteq_1(\mathsf{S},\mathsf{B}) \;\equiv\; \mathsf{R}\subseteq\mathsf{S}\wedge\mathsf{A}\supseteq\mathsf{B} \quad . \tag{8.18}$$

Of these, the first corresponds to

$$\forall(\mathsf{C}::\ \mathrm{wlp}.(\mathsf{R},\mathsf{A}).\mathsf{C}\supseteq\mathrm{wlp}.(\mathsf{S},\mathsf{B}).\mathsf{C}) \quad . \tag{8.19}$$

This order considers only the input-output relation of a program: i.e. it considers only partial correctness. The second one, formulated in terms of the *wlp* and *wp* functions, is:

$$\forall(\mathsf{C}::\ \mathrm{wlp}.(\mathsf{R},\mathsf{A}).\mathsf{C}\supseteq\mathrm{wlp}.(\mathsf{S},\mathsf{B}).\mathsf{C}) \;\wedge\; \forall(\mathsf{C}::\ \mathrm{wp}.(\mathsf{R},\mathsf{A}).\mathsf{C}\supseteq\mathrm{wp}.(\mathsf{S},\mathsf{B}).\mathsf{C}) \tag{8.20}$$

This is the refinement order of [49].

From the "refinement literature" [72, 71] the refinement order

$$(\mathsf{R},\mathsf{A})\sqsubseteq_2(\mathsf{S},\mathsf{B}) \;\equiv\; \forall(\mathsf{C}::\ \mathrm{wp}.(\mathsf{R},\mathsf{A}).\mathsf{C}\supseteq\mathrm{wp}.(\mathsf{S},\mathsf{B}).\mathsf{C}) \quad , \tag{8.21}$$

is known. Expressed in terms of relations it is:

$$(\mathsf{R},\mathsf{A})\sqsubseteq_2(\mathsf{S},\mathsf{B}) \equiv \mathsf{B}\circ\mathsf{R}\subseteq\mathsf{S}\wedge\mathsf{A}\supseteq\mathsf{B} \quad . \tag{8.22}$$

(In the proofs of the equality of (8.17) and (8.19), of (8.18) and (8.20) and finally of (8.21) and (8.22), the axiom of extensionality has to be used.)

It is not difficult to check that all operators $[\!]$, $\diamond$ and ";" are monotone with respect to the orders $\sqsubseteq_0$ and $\sqsubseteq_1$. It is also the case that $[\!]$ and ";" are monotone with respect to $\sqsubseteq_2$. However, the fair choice operator is not monotone with respect to $\sqsubseteq_2$. We have $\mathsf{skip} \sqsubseteq_2 \mathsf{abort}$, but:

$$
\begin{array}{ll}
 & (\mathsf{skip} \diamond \mathsf{miracle}) \ \sqsubseteq_2 \ (\mathsf{abort} \diamond \mathsf{miracle}) \\
\equiv & \qquad \{ \quad \text{definitions: section 4.1 ; definition 8.15} \quad \} \\
 & \mathsf{skip} \ \sqsubseteq_2 \ \mathsf{miracle} \\
\equiv & \qquad \{ \quad \text{definition (8.18)} \quad \} \\
 & \mathrm{I} \subseteq \perp\!\!\perp \\
\equiv & \qquad \{ \quad \text{Cone rule: } \mathrm{I} \neq \perp\!\!\perp \quad \} \\
 & \text{false}
\end{array}
$$

A refinement order must have the property that all operators on programs are monotone with respect to it. This monotonicity property is needed because it allows the refinement of a program by refining its subprograms. Therefore, the order $\sqsubseteq_2$ is not admissible for programs involving the fair choice operator. The order $\sqsubseteq_0$ is mainly of interest if one wants to consider only partial correctness. Because the model of programs from this chapter is designed with the intention to be able to study total correctness, the appropriate refinement order is $\sqsubseteq_1$. Note that $\sqsubseteq_1$, being the order on a product of two complete lattices, is a complete lattice. The top element of this lattice is $\mathsf{chaos}$, the bottom element $\mathsf{miracle}$ and the join operator the operator $[\!]$ .

The second type of order on programs comprises the approximation orders. Approximation orders are used to define recursive programs as least fixed points of monotone program transformers. Therefore, one traditionally requires an approximation order to be a complete partial order (a CPO) because this guarantees the existence of least fixed points of monotone functions. Furthermore, the monotone program transformers one needs to consider are built from the operators on programs. So, an approximation order must be monotone in all program constructors. A candidate approximation order is:

$$
(\mathsf{R},\mathsf{A}) \leq (\mathsf{S},\mathsf{B}) \quad \equiv \quad \mathsf{R} \subseteq \mathsf{S} \ \wedge \ \mathsf{A} \subseteq \mathsf{B} \quad . \tag{8.23}
$$

This order is a CPO. It is even a complete lattice (because it is a product order on the product of two complete lattices) with $\mathsf{abort}$ as bottom, $\mathsf{havoc}$ as top and the fair choice operator as join. However, the sequential composition operator is not monotone with respect to it. On the one hand, it is the case that $\mathsf{miracle} \leq \mathsf{skip}$. But on the other hand:

$$
\begin{array}{ll}
 & \mathsf{miracle;abort} \leq \mathsf{skip;abort} \\
\equiv & \qquad \{ \quad \text{definitions: section 4.1; definition 8.15; (2.40)} \quad \} \\
 & \mathsf{miracle} \leq \mathsf{abort}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definition 8.23 } \} \\
& \mathsf{I} \subseteq \perp\!\!\perp \\
\equiv & \{ \text{ Cone rule: } \mathsf{I} \neq \perp\!\!\perp \ \} \\
& \text{false}
\end{array}
$$

As a consequence it is not possible to use the Knaster-Tarski theorem to show that program transformers involving sequential composition have a least fixed point with respect to this order. Nevertheless we will see in the section on recursion that program transformers involving the operator ";" ( but not $\diamond$) can have a least fixed point with respect to $\leq$.

The standard way of constructing a more suitable approximation order out of $\leq$ is by restricting it in the following way.

$$
(\mathsf{R},\mathsf{A}) \preceq (\mathsf{S},\mathsf{B}) \quad \equiv \quad \mathsf{R} \subseteq \mathsf{S} \ \wedge \ \mathsf{A} \subseteq \mathsf{B} \ \wedge \ \mathsf{A} \circ \mathsf{S} \subseteq \mathsf{R} \tag{8.24}
$$

The order $\preceq$ is known as the *Egli-Milner* order. Expressed in terms of *wlp* and *wp* it is also known under the name *Nelson* order.

$$
\begin{array}{lll}
(\mathsf{R},\mathsf{A}) \preceq (\mathsf{S},\mathsf{B}) & \equiv & \forall(\mathsf{C} :: \ \mathrm{wlp}.(\mathsf{S},\mathsf{B}).\mathsf{C} \subseteq \mathrm{wlp}.(\mathsf{R},\mathsf{A}).\mathsf{C}) \\
& & \wedge \forall(\mathsf{C} :: \ \mathrm{wp}.(\mathsf{R},\mathsf{A}).\mathsf{C} \subseteq \mathrm{wp}.(\mathsf{S},\mathsf{B}).\mathsf{C})
\end{array}
$$

(Again: extensionality has to be used to show that the two formulations are equivalent)

It is not immediately clear that the relation $\preceq$ is a CPO and, more seriously, it is not monotone with respect to all operators. Specifically we have $\mathsf{abort} \preceq \mathsf{skip}$, but:

$$
\begin{array}{ll}
& \mathsf{abort} \diamond \mathsf{miracle} \preceq \mathsf{skip} \diamond \mathsf{miracle} \\
\equiv & \{ \text{ definitions: section 4.1; definition 8.15 } \} \\
& \mathsf{miracle} \preceq \mathsf{skip} \\
\equiv & \{ \ (8.24) \ \} \\
& \mathsf{I} \subseteq \perp\!\!\perp \\
\equiv & \{ \text{ Cone rule: } \mathsf{I} \neq \perp\!\!\perp \ \} \\
& \text{false}
\end{array}
$$

So, it is impossible for us to define a recursive program as a least fixed point under the Egli-Milner order, and therefore we need another technique.

## 8.5 Recursion

As we have remarked before, the standard approach to recursion is as follows. Consider a CPO, $\trianglelefteq$, on programs and a $\trianglelefteq$-monotone program transformer. The program defined by such a program transformer is then the least fixed point under the $\trianglelefteq$ order. The way we propose to tackle recursion differs from this approach in that we do not give an order with respect to which all program transformers are monotone. Instead we restrict our attention to the program transformers $\mathcal{F}$ satisfying the following definition.

**Definition 8.25 (Normal program transformers)** Program transformer $\mathcal{F}$ is called *normal* iff there is a function $\mathsf{F}$, from relations to relations, and a binary operator $\oplus$, from programs to monotypes, such that

$$\mathcal{F}.(\mathsf{R},\mathsf{A}) = (\mathsf{F.R},\mathsf{R}\oplus\mathsf{A})$$

It is furthermore required that $\mathsf{F}$ is $\subseteq$-monotone and that the function $(\mathsf{A}\mapsto\mathsf{R}\oplus\mathsf{A})$ is $\subseteq$-monotone for all relations $\mathsf{R}$.
□

It is straightforward to check that program transformers built from constants, the identity function, sequential composition and the choice operators are all normal. To give the intuition behind the notion normal: one should interpret $\mathcal{F}.\mathsf{r}$ as a sequential program which contains the statement $\mathsf{r}$. That $\mathcal{F}$ is normal tells us three things.

- First, the i-o relation of $\mathcal{F}.\mathsf{r}$ does not depend on the safe monotype of $\mathsf{r}$. That is, if statement $\mathsf{r}$ is replaced by a statement with the same i-o relation but with different termination properties, the i-o relation of the program does not change.
- Second: if $\mathsf{r}$ is replaced by a statement with a larger i-o relation, the i-o relation of the program tends to grow.
- And third: if $\mathsf{r}$ is replaced by a statement that terminates more often, the program as a whole tends to terminate more often.

These requirements are, from an operational point of view, rather reasonable. In fact a fourth property suggests itself:

- If $\mathsf{r}$ is replaced by a statement with a greater i-o relation, the program will be more likely to diverge, i.e. its safe monotype tends to decrease.

(As a matter of fact the last three properties are equivalent to $\sqsubseteq_1$-monotonicity of $\mathcal{F}$ .)
If a normal program transformer also has the fourth property it is called paranormal.

**Definition 8.26 (Paranormal program transformers)** A program transformer is called *paranormal* iff it is normal and, furthermore, the operator $\oplus$ has the property that the function $(\mathsf{R}\mapsto\mathsf{R}\oplus\mathsf{A})$ is $\subseteq$-antimonotone for all monotypes $\mathsf{A}$.
□

Again, it is a routine matter to check that program transformers built from constants, the identity function, sequential composition and the choice operators are all paranormal.

The problem of recursion is then: which fixed point do we assign to a normal program transformer $\mathcal{F}$? The i-o relation component suggests itself: recall that the relations form a complete lattice, so the least fixed point of the monotone function $\mathsf{F}$ exists. So we can try to construct a fixed point $\kappa\mathcal{F}$ with as property $\kappa\mathcal{F} = (\mu\mathsf{F},\mathsf{A})$, for some $\mathsf{A}$. Let us now see what the constraints on $\mathsf{A}$ are. First of all $\kappa\mathcal{F}$ should be a fixed point.

$$
\begin{array}{ll}
 & \mathcal{F}\,.\,\kappa\mathcal{F} = \kappa\mathcal{F} \\
\equiv & \qquad \{\ \ \mathcal{F} \text{ is normal; definition of } \kappa\mathcal{F}\ \ \} \\
 & (\mathsf{F}\,.\,\mu\mathsf{F}\ ,\ \mu\mathsf{F} \oplus \mathsf{A}) = (\mu\mathsf{F}\,,\mathsf{A}) \\
\equiv & \qquad \{\ \ \mathsf{F}\,.\,\mu\mathsf{F} = \mu\mathsf{F}\ \ \} \\
 & \mu\mathsf{F} \oplus \mathsf{A} = \mathsf{A}
\end{array}
$$

So, $\mathsf{A}$ is a fixed point of the function $(\mathsf{B} \mapsto \mu\mathsf{F} \oplus \mathsf{B})$. Because $\mathcal{F}$ is normal this function is monotone and therefore has fixed points. Let us take "the least safe solution" $\mu(\mathsf{B} \mapsto \mu\mathsf{F} \oplus \mathsf{B})$.

**Definition 8.27 (Recursion)** Let $\mathcal{F}$ be a normal program transformer with $\mathsf{F}$ and $\oplus$ as in definition 8.25 then the program $\kappa\mathcal{F}$ defined by $\mathcal{F}$ is:

$$\kappa\mathcal{F} = (\mu\mathsf{F}\,, \mu(\mathsf{B} \mapsto \mu\mathsf{F} \oplus \mathsf{B}))$$

□

In this way recursion is very easy to handle. There are no complications caused by so-called unbounded nondeterminism, this notion not even entering the picture. (One of the reasons for this is that it is very hard to express that property using relations.) More importantly, it is unnecessary to restrict oneself to Egli-Milner-monotone program constructs.

The question then of course arises whether this approach is compatible with other treatments of recursion if one does make the restriction to Egli-Milner-monotone constructs. The (positive) answer to that question will be given in the next section, but first we give as an illustration, a program involving recursion and a non Egli-Milner-monotone (but paranormal) program transformer. Consider for integer variable $\mathsf{a}$ the program transformer given by:

$$
\begin{array}{lcl}
\mathcal{F}.x & = & \mathsf{a} := \mathsf{a}{+}1\ ;\ x\ ;\ \mathsf{a} := \mathsf{a}{+}1 \\
 & & \diamond\ !(\mathsf{a} = 0) \\
 & & \diamond\ \mathsf{a} := \mathsf{a}{-}1\ ;\ x\ ;\ \mathsf{a} := \mathsf{a}{-}1
\end{array}
$$

So, we focus our attention on relations on the integers. The program $\mathsf{a} := \mathsf{a}{+}1$ has as i-o relation the successor relation *succ*: $\mathsf{i}\langle \text{succ}\rangle \mathsf{j} \equiv \mathsf{i}{+}1 = \mathsf{j}$ and diverges nowhere and hence: $\mathsf{a} := \mathsf{a}{+}1 = (\text{succ}\,, \mathrm{I})$. Similarly $\mathsf{a} := \mathsf{a}{-}1 = (\text{pred}\,, \mathrm{I})$, where *pred* is the predecessor relation: $\mathsf{i}\langle \text{pred}\rangle \mathsf{j} \equiv \mathsf{i}{-}1 = \mathsf{j}$. Finally we have $!(\mathsf{a} = 0) = (\mathsf{Z}\,, \mathsf{Z})$ where $\mathsf{Z}$ is the identity relation on the one element set $\{0\}$. Let us take for $\mathsf{x}$ the program $(\mathsf{X}\,, \mathsf{E})$, then the righthand side "translates" to

$$
\begin{array}{ll}
 & (\text{succ}\,,\mathrm{I});(\mathsf{X}\,,\mathsf{E});(\text{succ}\,,\mathrm{I}) \diamond (\mathsf{Z}\,,\mathsf{Z}) \diamond (\text{pred}\,,\mathrm{I});(\mathsf{X}\,,\mathsf{E});(\text{pred}\,,\mathrm{I}) \\
= & \qquad \{\ \ \text{definitions and calculus}\ \ \} \\
 & (\text{succ} \circ \mathsf{X} \circ \text{succ} \cup \mathsf{Z} \cup \text{pred} \circ \mathsf{X} \circ \text{pred}\quad ,\quad \mathsf{E}\!\not/\text{succ} \cup \mathsf{Z} \cup \mathsf{E}\!\not/\text{pred})
\end{array}
$$

Thus, $\mathcal{F}.(X\,,E) = (F.X\,,X{\oplus}E)$ where $F$ and $\oplus$ satisfy:

$$F.X \;=\; \text{succ}\circ X\circ\text{succ} \;\cup\; Z \;\cup\; \text{pred}\circ X\circ\text{pred} \quad \text{, and} \tag{8.28}$$

$$X{\oplus}E \;=\; E/\!\!\!\!\!\text{succ} \cup Z \cup E/\!\!\!\!\!\text{pred} \quad . \tag{8.29}$$

It is easy to see that $\mathcal{F}$ is normal. It is equally easy to see that it is not Egli-Milner-monotone. While $\mathsf{abort} \preceq \mathsf{skip}$, it is not the case that $\mathcal{F}.\mathsf{abort} \preceq \mathcal{F}.\mathsf{skip}$.

$$
\begin{array}{ll}
 & \mathcal{F}.\mathsf{abort} \preceq \mathcal{F}.\mathsf{skip} \\
\equiv & \qquad \{ \;\; \text{calculus or interpretation} \;\; \} \\
 & !(a = 0) \;\preceq\; a := a{+}1\,;\,a := a{+}1 \;\Diamond\; !(a = 0) \;\Diamond\; a := a{-}1\,;\,a := a{-}1 \\
\equiv & \qquad \{ \;\; \text{definitions; definition 8.15.} \;\; \} \\
 & (Z\,,Z) \preceq (\text{succ}\circ\text{succ} \;\cup\; Z \;\cup\; \text{pred}\circ\text{pred}\;,\,\mathrm{I}) \\
\Rightarrow & \qquad \{ \;\; (8.24) \;\; \} \\
 & Z\circ(\text{succ}\circ\text{succ} \;\cup\; Z \;\cup\; \text{pred}\circ\text{pred}) \subseteq Z \\
\Rightarrow & \qquad \{ \;\; \text{calculus} \;\; \} \\
 & Z\circ\text{succ}\circ\text{succ} \subseteq Z \\
\equiv & \qquad \{ \;\; \text{interpretation} \;\; \} \\
 & \text{false}
\end{array}
$$

What then is $\kappa\mathcal{F}$? Although this is messy, we can find $\mu\mathcal{F}$ using the Kleene construction. First note that for $n \geq 0$: $F^n.\perp\!\!\perp = \{(i\,,-i) \mid -n \leq i < n\}$. Hence: $\cup(n :: F^n.\perp\!\!\perp) = \{(i\,,-i) \mid i{\in}\mathbb{Z}\}$. And because this relation is a fixed point of $F$ it is the least fixed point. For the second component we proceed in a similar way. We have to construct the least fixed point of the function $(E \mapsto X{\oplus}E)$ i.e. the least fixed point of the function defined by $g.E = E/\!\!\!\!\!\text{succ}{\cup}Z{\cup}E/\!\!\!\!\!\text{pred}$. Using the interpretation of the monotype factor it is not difficult to show that for all $n \geq 0$: $g^n.\perp\!\!\perp = \{(i\,,i) \mid -n \leq i < n\}$. So we have $\cup(n :: F^n.\perp\!\!\perp) = \mathrm{I}$. And again this is a fixed point of $g$, as easily is checked, so it is the least fixed point. Now we have: $\kappa\mathcal{F} = (\{(i\,,-i) \mid i{\in}\mathbb{Z}\}\,,\mathrm{I}) = a := -a$.

This is also what one expects from operational considerations: If the program is ever to terminate on input $n$, it has to “pass through zero”. Zero is reached by either subtracting or adding one $n$ times, and in the process of returning after zero has been reached these addition or subtraction steps are repeated. Of course, it is very hard to reason about such a program in an operational way, because of the exploding number of “forks” during the execution. This explosion is also the reason that although the program is equal to $a := -a$, it is a very inefficient implementation of negation.

## 8.6 The restriction to Egli-Milner-monotony

In this section we study the question whether the definition of recursion from the previous section is compatible with existing notions. The main result is the following.

**Theorem 8.30** If $\mathcal{F}$ is a paranormal program transformer that is also Egli-Milner-monotone, then all fixed points $x$ of $\mathcal{F}$ satisfy:

$$\kappa\mathcal{F} \leq x \quad \text{and} \quad \kappa\mathcal{F} \preceq x \ .$$

□

This result means that a paranormal $\preceq$-monotone function has $\kappa\mathcal{F}$ as a least $\leq$-fixed point, even if the function is not $\leq$-monotone. Furthermore such a function has $\kappa\mathcal{F}$ as a least $\preceq$-fixed point (but, as will become evident, we do not need to know that $\preceq$ is a CPO). So, when we restrict our attention to paranormal, Egli-Milner monotone program transformers $\mathcal{F}$, the fixed point $\kappa\mathcal{F}$ coincides with the least Egli-Milner fixed point. In other words, in that case our approach to recursion coincides with the standard one. Unfortunately, the proof of theorem 8.30 is rather complicated. It is performed entirely in the relational calculus and this helps to reduce its complexity. As a first stepping stone to the main result we prove a lemma.

**Lemma 8.31** If $\mathcal{F}$ is a paranormal program transformer with $F$ and $\oplus$ as in definition 8.25 and if $\mathcal{F}$ is also Egli-Milner monotone, then we have for all relations $S$ and monotypes $A$:

$$S \oplus A = (A \circ S) \oplus A \quad \text{and} \quad S \oplus A \circ F.S \;=\; S \oplus A \circ F.(A \circ S) \ .$$

**Proof** Function $\mathcal{F}$ is Egli-Milner-monotone means that for any $R$, $S$, $A$ and $B$ it is the case that

$$R \subseteq S \wedge A \subseteq B \wedge A \circ S \subseteq R \;\Rightarrow\; F.R \subseteq F.S \;\wedge\; R \oplus A \subseteq S \oplus B \;\wedge\; R \oplus A \circ F.S \subseteq F.R$$

By instantiating $R := A \circ S$ and $B := A$ the antecedent is true. So we have:

$$F.(A \circ S) \subseteq F.S \quad , \tag{8.32}$$

$$(A \circ S) \oplus A \subseteq S \oplus A \quad \text{and} \tag{8.33}$$

$$(A \circ S) \oplus A \circ F.S \subseteq F.(A \circ S) \quad . \tag{8.34}$$

Observing that $A \subseteq I$, hence $A \circ S \subseteq S$ and using that $\oplus$ is $\subseteq$-antimonotone in its first argument, we see that (8.33) can be strengthened to an equality. This gives us the first conjunct of the lemma. For the second conjunct we do the following calculation.

$$\begin{array}{ll}
& S{\oplus}A \circ F.S \;=\; S{\oplus}A \circ F.(A \circ S) \\
\equiv & \quad \{ \;\; (8.32);\ \text{monotonicity of} \circ \;\; \} \\
& S{\oplus}A \circ F.S \subseteq S{\oplus}A \circ F.(A \circ S) \\
\Leftarrow & \quad \{ \;\; \text{monotonicity of} \circ;\ S{\oplus}A \text{ is a monotype; for monotypes: } B \circ B = B \;\; \} \\
& S{\oplus}A \circ F.S \subseteq F.(A \circ S) \\
\equiv & \quad \{ \;\; \text{The already proven first conjunct.} \;\; \} \\
& (A \circ S){\oplus}A \circ F.S \subseteq F.(A \circ S) \\
\equiv & \quad \{ \;\; (8.34) \;\; \} \\
& \text{true}
\end{array}$$

□

It is possible to give an operational interpretation to this lemma. However, we will refrain from doing so, because it is rather difficult to formulate and not very illuminating. Now we do the next step in the proof of the main result.

**Lemma 8.35** For $\mathcal{F}$ a function satisfying the conditions of lemma 8.31 and $R$ a relation such that $R = F.R$:

$$\mu(A \mapsto R{\oplus}A) \circ R \subseteq \mu F \quad \text{and}$$

$$\mu(A \mapsto \mu F \oplus A) = \mu(A \mapsto R{\oplus}A)$$

**Proof** The form of the first proof obligation invites the use of $\mu$-fusion.

$$\begin{array}{ll}
& \mu(A \mapsto R{\oplus}A) \circ R \subseteq \mu F \\
\Leftarrow & \quad \{ \;\; \mu\text{-fusion: } (\circ R) \text{ has an upper adjoint;} \\
& \quad\quad\;\; \text{for all monotypes } A \;\; \} \\
& R{\oplus}A \circ R \subseteq F.(A \circ R) \\
\equiv & \quad \{ \;\; \text{assumption: } R = F.R \;\; \} \\
& R{\oplus}A \circ F.R \subseteq F.(A \circ R) \\
\equiv & \quad \{ \;\; \text{lemma 8.31} \;\; \} \\
& R{\oplus}A \circ F.(A \circ R) \subseteq F.(A \circ R) \\
\equiv & \quad \{ \;\; R{\oplus}A \subseteq I \;\; \} \\
& \text{true}
\end{array}$$

The proof of the second conjunct is:

$$
\begin{array}{ll}
 & \mu(A \mapsto \mu F \oplus A) = \mu(A \mapsto R \oplus A) \\
\equiv & \quad \{ \quad \mu F \subseteq R\text{; anti-monotonicity of } \oplus A\text{; monotonicity of } \mu \quad \} \\
 & \mu(A \mapsto \mu F \oplus A) \subseteq \mu(A \mapsto R \oplus A) \\
\Leftarrow & \quad \{ \quad \text{Knaster-Tarski} \quad \} \\
 & \mu F \oplus \mu(A \mapsto R \oplus A) \subseteq \mu(A \mapsto R \oplus A) \\
\equiv & \quad \{ \quad \text{unfold the fixed point} \quad \} \\
 & \mu F \oplus \mu(A \mapsto R \oplus A) \subseteq R \oplus \mu(A \mapsto R \oplus A) \\
\equiv & \quad \{ \quad \text{lemma 8.31} \quad \} \\
 & \mu F \oplus \mu(A \mapsto R \oplus A) \subseteq (\mu(A \mapsto R \oplus A) \circ R) \oplus \mu(A \mapsto R \oplus A) \\
\Leftarrow & \quad \{ \quad \text{anti-monotonicity of } \oplus \quad \} \\
 & \mu(A \mapsto R \oplus A) \circ R \subseteq \mu F \\
\equiv & \quad \{ \quad \text{The already proven first conjunct of lemma 8.35} \quad \} \\
 & \text{true}
\end{array}
$$

□

Using this last lemma we can now easily prove the main result. Assume that $\mathcal{F}$ is an Egli-Milner-monotone. Assume, furthermore, that $F$ and $\oplus$ witness the fact that $\mathcal{F}$ is paranormal. For the first conjunct of theorem 8.30 we calculate:

$$
\begin{array}{ll}
 & \kappa\mathcal{F} \leq (R, B) \\
\equiv & \quad \{ \quad \text{definition 8.27} \quad \} \\
 & \mu F \subseteq R \wedge \mu(A \mapsto \mu F \oplus A) \subseteq B \\
\Leftarrow & \quad \{ \quad \text{Knaster-Tarski; lemma 8.35} \quad \} \\
 & F.R = R \wedge \mu(A \mapsto R \oplus A) \subseteq B \\
\Leftarrow & \quad \{ \quad \text{Knaster-Tarski} \quad \} \\
 & F.R = R \wedge R \oplus B = B \\
\equiv & \quad \{ \quad \text{assumptions on } \mathcal{F} \quad \} \\
 & \mathcal{F}.(R, B) = (R, B)
\end{array}
$$

The second conjunct is yet easier to show.

$$
\begin{array}{ll}
 & \kappa\mathcal{F} \preceq (R, B) \\
\equiv & \quad \{ \quad (8.24); (8.23) \quad \} \\
 & \kappa\mathcal{F} \leq (R, B) \ \wedge \ \mu(A \mapsto \mu F \oplus A) \circ R \subseteq \mu F
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ above calculation; lemma 8.35 } \} \\
& \mathcal{F}.(\mathsf{R}\,,\mathsf{B}) = (\mathsf{R}\,,\mathsf{B}) \wedge \mathsf{F.R} = \mathsf{R} \\
\equiv & \{ \text{ second conjunct is implied by first } \} \\
& \mathcal{F}.(\mathsf{R}\,,\mathsf{B}) = (\mathsf{R}\,,\mathsf{B})
\end{array}
$$

This concludes the proof of theorem 8.30.

## 8.7 Summary and conclusions

In this chapter we have presented a relational model of programs. By viewing a program as a pair consisting of a relation and a set it is possible to study both boundedly and unboundedly nondeterministic programs that may terminate or diverge on the same input. We discussed various ways of combining programs, including a so-called fair choice. Recursion is handled by taking fixed points (not necessarily least fixed points) of normal program transformers. The class of these functions is larger than the usual class of Egli-Milner-monotone functions. Finally we showed that our method of treating recursion is compatible with existing notions.

Some conclusions are to be drawn. First of all the relational calculus turned out to be an excellent tool for our purposes. This is mainly due to the use of the monotype factor and exploiting its defining Galois connection. Other treatments like [16] also use the same function but, because they do not use the Galois connection but use a closed form instead their treatment is more complicated. Another conclusion is that an "erratic" approach to program semantics is, in a way, better than a demonic or angelic approach. Erratic is better than angelic, because it allows the study of nonterminating programs. It is better than demonic because it allows constructions like the fair choice, which have in a certain sense an angelic behaviour: termination is guaranteed if termination of one of its components is guaranteed. A demonic approach does not allow such constructs: from a demonic point of view divergence is guaranteed if one of the components diverges.

A third conclusion is that it is possible to give a simple treatment of recursion. It does not seem to be true as Dijkstra claims in [35] and Nelson agrees in [73] that "...recursion is an order of magnitude more difficult than just repetition". One wonders why other treatments are more complicated, resorting to Egli-Milner monotony. Part of the answer is perhaps that traditionally there is a fixation on least fixed points: one wants to define a recursive program as a least fixed point of a monotone function, without separating the concerns of termination and input-output behaviour. Another part of the answer (which is not totally disjoint from the first part) might be the traditional stress on total correctness and the function *wp* . One is primarily interested in the wp.$s$ of program $s$ and for some reason seldom exploits the termination law (8.4): one has total knowledge of the function wp.$s$ if one knows the function wlp.$s$ and the set wp.$s$.I. For instance Dijkstra and Scholten remark in [37] when they treat the repetition: "Because it is more convenient ...we shall define wlp.DO and wp.DO independently", without providing any evidence why this is more convenient. In fact we claim that it is not more convenient: it gives a better separation of concerns and an easier treatment to construct the function wlp.s (i.e. the input-output behaviour of program s) first and then, using the wlp.s , find the set wp.s.I. If necessary the termination law can then be used to construct the function wp.s .

One is tempted to conclude that it is unwise to have redundant information in a definition (such as is the case in the *wlp*-*wp*-pair) because it might be a hindrance to the easy development of a theory.

Finally we want to remark that the operator $\diamond$ deserves more study. It can be handy in the specification of searching problems. Another use is the specification of parsers. The construct tends to lead to inefficient programs (although it is easy to implement it on a parallel machine), so we need refinement rules to transform it away. One such a rule is:

$$!b;r \diamond !(\neg b);r \;=\; !b;r \,[\!]\, !(\neg b);r$$

Another interesting point is the question of the application of the operator in Unity [28]. A Unity program is (if we forget about fairness) basically a program of the form:

$$\mathsf{do}\ \text{true} \rightarrow [\!](i ::\ s_i)\ \mathsf{od} \quad ,$$

where $\{s_i\}$ is a set of statements that are guaranteed to terminate. What happens now if we drop the termination requirement and change the program into:

$$\mathsf{do}\ \text{true} \rightarrow \diamond(i ::\ s_i)\ \mathsf{od} \quad ? \tag{8.36}$$

For instance, assume that the statement $x := x\ \underline{\mathsf{div}}\ y$ is equal to $\mathsf{abort}$ when started in a state satisfying $y = 0$. Then a program consisting of the statements $x := x\ \underline{\mathsf{div}}\ y$ and $y := 0$ would not be an admissible Unity program, because then the aborting statement causes the whole program to abort. On the other hand if one interprets it using (8.36) there does not seem to be any problem.

# Chapter 9

# Epilogue

In this thesis, our primary goal has been to show how core concepts in programming, like the notions of datatype, specification, program, and termination, can be captured compactly in the relational framework. Although important, compactness was, nevertheless, not our main concern. Our main concern was to formalise the concepts in such a way that one can work with them; that is to say that they are useful for programming.

Programs can be complicated and programming difficult. We have tried to isolate the more straightforward phases of the design of programs. To be able to do this we had to formalise the concepts in such a way that the typical structure of a design becomes apparent. Complicated mappings of programs to yet more complicated models can not help to reveal this structure. What is needed is a simple view on programs. Relations are in this respect an excellent means to understand programs on an abstract level: they are not too complicated and the calculus of relations can be used to reason about them.

An example of a straightforward stage is the design of the termination argument in many repetitions. Often the termination argument dictates the structure of the repetition body: it is the first step towards a solution. Typically, however, this design process is only applied to programs involving very simple data structures, arrays being the most complex.

Ideally, the same techniques should also apply to the construction of programs with recursion schemes different from that of the repetition, and working on more complicated data structures than arrays or lists: termination has to dictate the shape of the program. A number of standard schemes then becomes desirable. The identification of some standard schemes constitutes the reductivity theorems of chapter 6.

The mathematics of program construction has to be as simple as possible. The relatively simple point-free relation calculus lends itself very well for reasoning about programs on an abstract level. A relation can be seen as corresponding to a procedure where much of the internal structure is abstracted away. Therefore, relation calculus is useful to reason compactly on the level of recursion schemes, to study generally valid program transformations and to calculate the typical structure of a terminating program.

On the other hand, when the concrete structure of a procedure and the data the program acts upon have to be taken into account, pointwise reasoning is often more appropriate. This can be seen from the example programs given in this thesis. As soon as one considers more concrete programs the point-free expressions become messy and less, rather than more compact than their pointwise counterparts. This, of course, is because the properties of the points have to be coded by relations, and that is not what relations are suitable for.

It is possible to mimic pointwise reasoning in the relation calculus. But there is no compelling need to do so: there are excellent tools available for pointwise reasoning such as the predicate calculus. The important thing for a programmer to have is a suitable interface between pointwise and point-free reasoning.

In a similar way, it should be possible for a programmer to use different programming techniques in the course of the design of a program. But more importantly, one should be able to identify which techniques are most suitable in a particular stage of the design. Techniques from functional programming can be used in the design of imperative programs. For instance, a tail recursive function can be implemented with a repetition. On the other hand, the techniques of constructing repetitions boil down to constructing a recurrence relation for the repetition invariant. These techniques can also be applied to construct functional programs. The concepts presented here are meant to enable the investigation and discussion of such issues.

For instance, what we have found and shown is that, when programs are viewed on an abstract level, there is no major difference between imperative and functional programming (as long as functions are not passed as parameters to other functions). It is very well possible that on this level the differences with logic programming are minor as well and that the introduction of techniques from logic programming can also help in the design of imperative or functional programs.

There are additional reasons to investigate logic programming. First, it might very well be that the understanding of logic programs can increase considerably if such programs are seen as relations rather than predicates. For instance, so-called "logic variables" are just a means to express the composition of two relations. Second, the execution of logic programs involves backtracking. Therefore, the formalisation of termination of logic programs might lead to an interesting concept, in much the same way as formalisation of termination of imperative programs lead to reductivity.

And then there is the challenge of the infinite list programs and lazy evaluation, programs that do not necessarily terminate. Can these be modelled as relations? This problem was already mentioned by Malcolm in the concluding chapter of his thesis and has still to be resolved. However, there is evidence that relational methods are suitable for this. A program generating an infinite list does not terminate. The property that such a program should have instead is that all finite initial segments can be generated in finite time. So, such a program can also be seen as a relation between the input and all finite initial segments of an infinite list, rather than a relation between the input and the infinite list.

However, the real challenge is to put the theory of this thesis to work in the construction of programs, because that is what it was designed for: the choice of the concepts to be considered and the way in which they were formalised have been heavily inspired by our experiences in designing and explaining programs. Only by a balanced and well-considered combination of theory and practice can we hope to make program design simpler.

# Bibliography


[1] C.J. Aarts. Galois connections presented calculationally. Afstudeer verslag (Graduating Dissertation), Department of Computing Science, Eindhoven University of Technology, 1992.

[2] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans, and J. van der Woude. A relational theory of datatypes. Available via world-wide web at `http://www.win.tue.nl/win/cs/wp/papers`, September 1992.

[3] L. Bachmair and N. Dershowitz. Commutation, transformation and termination. In *Proceedings of the 8th Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 1986.

[4] R.J.R. Back and J. von Wright. A lattice-theoretical basis for a specification language. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 139–156, Berlin, 1989. Springer.

[5] R.J.R. Back and J. von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In W.-P. de Roever J.W. de Bakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[6] R.C. Backhouse. *Closure algorithms and the star-height problem of regular languages.* PhD thesis, University of London, 1975.

[7] R.C. Backhouse. *Program Construction and Verification.* Prentice-Hall International, 1986.

[8] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.

[9] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1*

*Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.

[10] R.C. Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15:161–186, 1975.

[11] R.C. Backhouse and P. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H.A. Partsch, and S.A. Schuman, editors, *Formal Program Development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar, Rio de Janeiro, Jan. 1992*, volume 755 of *LNCS*, pages 7–42. Springer-Verlag, 1993.

[12] R.C. Backhouse and J. van der Woude. Demonic operators and monotype factors. *Mathematical Structures in Computer Science*, 3(4):417–433, December 1993.

[13] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[14] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[15] Han Bäumer. On the use of relation algebra in the theory of reduction systems. In J.L.G. Dietz, editor, *CSN 92*, pages 54–64, Amsterdam, 1992. Centrum voor Wiskunde en Informatica.

[16] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoretical Computer Science*, 43:123–147, 1986.

[17] Richard S. Bird and Oege de Moor. Solving optimisation problems with catamorphisms. In R.S. Bird, C.C. Morgan, and J. C. P. Woodcock, editors, *2nd International Conference on the Mathematics of Program Construction*, volume 669, pages 45–66. Springer Verlag LNCS, June/July 1992.

[18] Richard S. Bird and Oege de Moor. From dynamic programming to greedy algorithms. In B. Möller, H.A. Partsch, and S.A. Schuman, editors, *Formal Program Development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar, Rio de Janeiro, Jan. 1992*, volume 755 of *LNCS*, pages 43–61. Springer-Verlag, 1993.

[19] Richard S. Bird and Oege de Moor. List partitions. *Formal Aspects of Computing*, 5(1):61–78, 1993.

[20] Richard S. Bird and Oege de Moor. *The Algebra of Programming.* To be published, Prentice-Hall International, 1996.

[21] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design.* Springer-Verlag, 1987. NATO ASI Series, vol. F36.

[22] R.S. Bird and L. Meertens. Two exercises found in a book on algorithmics. In L.G.L.T. Meertens, editor, *Program Specification and Transformations*, pages 451–457. Elsevier Science Publishers B.V., North Holland, 1987.

[23] Garrett Birkhoff. *Lattice Theory*, volume 25 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, Providence, Rhode Island, 3rd edition, 1967.

[24] M. Broy, R. Gnatz, and M. Wirsing. Semantics of nondeterministic and noncontinuous constructs. In F.L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 553–592, Berlin, 1979. Internat. Summer School, Marktoberdorf, Springer.

[25] M. Broy and G. Nelson. Adding fair choice to Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 16:924–938, 1994.

[26] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[27] R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[28] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[29] W. Chen and J.T. Udding. Program inversion: more than fun. Technical report, Department of Computing Science, University of Groningen, 1989.

[30] Edward Cohen. *Programming in the 1990s: an introduction to the calculation of programs*. Texts and monographs in computer science. Springer, Berlin, 1990.

[31] J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[32] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *Proceedings of EUROCAL 85*, pages 151–184. Springer-Verlag, LNCS 203, 1985.

[33] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, first edition, 1990.

[34] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In M. Nivat, editor, *Automata, Languages and Programming 1972*, pages 167–196. North Holland, 1973.

[35] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[36] E.W. Dijkstra and W.H.J. Feijen. *Een Methode van Programmeren.* Academic Service, Den Haag, 1984. Also available as *A Method of Programming*, Addison-Wesley, Reading, Mass., 1988.

[37] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.

[38] R.M. Dijkstra. Relational calculus and relational program semantics. Technical Report CS-R9408, Department of Computing Science, University of Groningen, 1994. (Originally written as a Masters thesis in Mathematics at the Universität zu Köln).

[39] R.P. Dilworth. Non-commutative residuated lattices. *Transactions of the American Mathematical Society*, 46:426–444, 1939.

[40] H. Doornbos. A relational model of programs without the restriction to Egli-Milner constructs. In E.-R. Olderog, editor, *Proc. IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 357–376. International Federation of Information Processing, June 1994.

[41] W.H.J. Feijen and A.J.M. van Gasteren. An introduction to the relational calculus. In W.H.J. Feijen and A.J.M. van Gasteren, editors, *Van oude machines en nieuwe rekenwijzen*, pages 57–81. Academic service, Schoonhoven, 1991.

[42] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, The Netherlands, 1992.

[43] Peter Freyd, Paul Hoogendijk, and Oege de Moor. Membership of datatypes. Unpublished draft.

[44] P.J. Freyd and A. Scedrov. *Categories, Allegories.* North-Holland, 1990.

[45] A. Geser. *Relative Termination.* PhD thesis, University of Passau, 1990.

[46] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[47] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 497–604. Reidel, 1984.

[48] W.H. Hesselink. Modalities of nondeterminism. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business.* Springer-Verlag, 1992.

[49] W.H. Hesselink. *Programs, Recursion and Unbounded Choice.* Cambridge University Press, 1992.

[50] Wim H. Hesselink. Lr-parsing derived. *Science of Computer Programming*, 19:171–196, 1992.

[51] Wim. H. Hesselink. Angelic termination in Dijkstra's calculus. In Bernhard Möller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 322–334. Springer, 1995.

[52] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[53] C.A.R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, 9:51–84, 217–252, 1986.

[54] Paul Hoogendijk. *A generic theory of datatypes*. PhD thesis, Eindhoven University of Technology, 1996. To appear.

[55] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, 1980.

[56] K. Iverson. *A Programming Language*. John Wiley & Sons, New York, 1962.

[57] Anne Kaldewaij. *Programming: the derivation of algorithms*. Prentice Hall international series in computer science. Prentice Hall, London, 1990.

[58] J.W. Klop. Term rewriting systems: a tutorial. *Bulletin of the EATCS*, 32:143–182, 1987.

[59] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[60] Roger D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. Department of Mathematics, Iowa State University, Ames, IOWA 50011, USA, May 1990.

[61] Roger D. Maddux. A working relational model: The derivation of the Dijkstra-Scholten predicate transformer semantics from Tarski's axioms for the Peirce-Schröder calculus of relations. Department of Mathematics, Iowa State University, Ames, Iowa 50011, USA, September 1992.

[62] G. Malcolm. *Algebraic data types and program transformation*. PhD thesis, Groningen University, 1990.

[63] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, October 1990.

[64] P. Martin-Löf. An intuistionistic theory of types. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118. North-Holland, 1975.

[65] Eindhoven University of Technology Mathematics of Program Construction Group. Fixed point calculus. *Information Processing Letters*, 53(3):131–136, February 1995.

[66] John Mc Carthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1963.

[67] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[68] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[69] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[70] Aart Middeldorp. *Modular properties of term rewriting systems*. PhD thesis, Vrije Universiteit Amsterdam, 1990.

[71] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[72] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Program.*, pages 287–306, 1987.

[73] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Languages and Systems*, 11:517–561, 1989.

[74] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Math.*, 43(2):223–243, 1942.

[75] Thanh Tung Nguyen. A relational model of nondeterministic programs. *International J. of Foundations of Computer Science*, 2(2):101–131, June 1991.

[76] F.J. Rietman. A note on extensionality. In J. van Leeuwen, editor, *Proceedings Computer Science in the Netherlands 91*, pages 468–483, 1991.

[77] Frans Johan Rietman. *A Relational Calculus for the Design of Distributed Algorithms*. PhD thesis, Department of Computing Science, Utrecht University, 1995.

[78] J. Riguet. Relations binaires, fermetures, correspondances de Galois. *Bulletin de la Société Mathématique de France*, 76:114–155, 1948.

[79] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. Assoc. Comp. Mach.*, 13(1):158–169, January 1966.

[80] G. Schmidt and T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg, 1993.

[81] Doaitse Swierstra and Oege de Moor. Virtual data structures. In Helmut Partsch, Bernhard Möller, and Steve Schuman, editors, *Formal Program Development*, volume 755 of *LNCS*, pages 355–371. Springer-Verlag, 1993.

[82] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

[83] A. Tarski. A fixed point theorem and its applications. *Pacific J. Math.*, pages 285–309, 1955.

[84] Jaap van der Woude. Calculations with relations, an example. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J Misra, editors, *Beauty is our Business*. Springer-Verlag, 1990.

# Index