The design of functional programs: a calculational approach

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus,
prof. ir. M. Tels,
voor een commissie aangewezen
door het College van Dekanen
in het openbaar te verdedigen op
dinsdag 19 december 1989 te 14.00 uur

door

ROBERT RICHARD HOOGERWOORD

geboren te Gorinchem

Dit proefschrift is goedgekeurd
door de promotoren

prof. dr. M. Rem
en
prof. dr. F.E.J. Kruseman Aretz

Contents

0 Introduction

0.0 The subject of this monograph

By now, it is well-known that there is only one way to establish the correctness of a computer program, namely by rigorous mathematical proof. Acceptance of this fact leaves room for two possible approaches to programming.

In the first approach, a program is constructed first and its correctness is proved afterwards. This approach has two drawbacks. First, for an arbitrary program it may be very difficult, if not impossible, to prove its correctness. Second, this approach sheds no light on the methodological question *how* programs are to be designed.

In the second approach, advocated and developed by E.W. Dijkstra [Dij0][Dij3] and others, the program and its proof of correctness are constructed simultaneously. Here, the obligation to construct a proof of correctness is used as a guiding principle for the design of the program. When applied rigorously, this approach can never give rise to incorrect programs; the worst thing that can happen is that no program is constructed at all because the problem is too difficult.

The latter approach turns out to be very effective. In the last 10 years, say, it has given rise to a *calculational style* of programming: programs are *derived* from their specifications by means of (chunks of) formula manipulation. These formal derivations take over the role of correctness proofs: programs are now correct by virtue of the way they have been constructed, which obviates the need for a posteriori correctness proofs.

Here we must add that, actually, there is no such thing as the correctness of a program; we can only speak of the correctness of a program with respect to a given specification: correctness means that the program satisfies the specification. Mathematically speaking, each pair (specification , program) represents a potential theorem requiring proof. This implies that both specification and program must be expressed in a strictly formal notation. Programming in a calculational way can then be defined as transforming the specification, by formal manipulations, into a program satisfying it.

In 1985 we started to investigate to what extent functional programs can be designed in a calculational way. This should be possible because functional-program notations carry less operational connotations than their sequential counterparts do: functional-program notations more resemble "ordinary" mathematical formalisms than sequential-program notations do. Moreover, we asked ourselves whether the two ways of programming are really different: they might very well turn out to have more in common than one would expect at first sight.

The results of this research are laid down in this monograph. This study is about programming, as a design activity; it is not about programming languages, formal semantics included, nor about implementations. This implies that we discuss semantics and implementations only as far as needed for our purpose, namely the formulation of a set of rules for designing programs.

The programming style presented in this monograph bears a strong resemblance with the *transformational style* developed by R.M. Burstall and J. Darlington [Bur][Dar0]; yet, there are a few, small but essential, differences. First, in the Burstall/Darlington style the starting point of a derivation always is a program. The goal of the transformation process is to obtain an equivalent program that, in some aspects such as efficiency, is better than the program one starts with. We prefer, however, to start with specifications that need not be programs. Second, the Burstall/Darlington system has been designed for mechanical program transformations. As a result, the set of transformation rules is rather limited and the resulting programs are partially correct only.

0.1 On the functional-program notation

Although this is not a study about programming languages, this does not mean that the notation used is irrelevant; on the contrary! Experience shows that program derivations are at least an order of magnitude longer than the actual code of the program derived. In order to keep the process of formula manipulation manageable, conciseness of the notation is of utmost importance [Gas]. We illustrate this by showing encodings of a, very simple, function definition in LISP and in our program notation. This function maps a nonempty list to its last element:

```
(last (lambda (x)
      (cond ( (equal (cdr x) nil) (car x) )
            (  t  (last (cdr x)) )
      )
     )
)   ,
```

```
last·(a;s)  =  ( s = [] → a
               ▯ s ≠ [] → last·s
               )   .
```

The LISP version is so baroque that it prohibits efficient formal manipulations. In view of this, it is no surprise that SASL, designed by D.A. Turner [Tur0], has been so successful and so inspiring: indeed, SASL is a very concise notation.

As a matter of fact, our own notation has been strongly inspired by SASL. Yet, we deliberately decided not to adopt SASL or any other existing program notation. We were not interested in the question "How to program in notation X?", for whatever notation X one likes. Our goal was to develop a calculational programming style that might be called "functional"; the program notation used should reflect this programming style. So, we expected that, as time went by, the program notation would evolve in harmony with our way of programming.

Any program notation is a mathematical formalism that also admits an operational interpretation. By their very nature, functional-program notations lend themselves very well to a clear separation of these two aspects. It has been quite a surprise to observe that so many researchers in this area ignore this distinction. The most marked example of this is the wide-spread use of the phrases *lazy language* and *lazy semantics* (sic!), which refer to the fact that the implementation of the notation requires lazy evaluation. We have tried to maintain the distinction between notation and operational interpretation as much as possible. Fortunately, this is not difficult at all.

We use an axiomatic characterisation of the program notation. This enables us to state exactly those properties of the notation we need, and as little more as possible. As a result, the program notation has not been defined completely. We have not even strived for completeness in the mathematical sense of the word; we do not care whether or not all true theorems in the system can be proved. We believe, with good reason, that the rules are sufficiently rich to be useful for programming.

0.2 The structure of this monograph

This monograph consists of three parts. In the first part, consisting of chapters 1,2,3, and 5, we present a formal definition of a functional-program notation and a theory for its use. Readers with some familiarity with SASL-like notations and with a main interest in program design may wish to skip these chapters. The main syntactic differences between our notation and SASL are the use of $\cdot$ ("dot") for function application, the use of brackets $[\![\cdots]\!]$ instead of **where** $\cdots$ for where-clauses, the way in which case analysis is denoted (section 2.5), the use of ; ("cons") instead of : for the list constructor, and the $\uparrow$ ("take") and $\downarrow$ ("drop") operators (section 5.1). Section 5.8 provides a summary of the most frequently used properties of the list operators.

In the second part, consisting of chapters 4 and 6, we present a number of programming techniques. The techniques presented in chapter 4 pertain to the use of recursion, generalisation, tupling and the use of additional parameters. They are elementary in the sense that they are simple and almost always applicable. In chapter 6 we discuss a number of techniques for designing programs in which lists play an important role. The use of the techniques is illustrated by means of small examples.

Finally, the third part consists of chapters 7 through 10. In each of these chapters we apply the programming techniques developed in the second part to derive programs for a programming problem. These chapters occur, more or less, in the order of increasing difficulty of the problems, but they are largely independent and can be read in any order.

0.3 Notational and mathematical conventions

We use *left-binding* and *right-binding* instead of the usual phrases *left-associative* and *right-associative* to denote parsing conventions for binary operators. An operator $\oplus$ is called left-binding if $x \oplus y \oplus z$ must be read as $(x \oplus y) \oplus z$ and $\oplus$ is called right-binding if $x \oplus y \oplus z$ must be read as $x \oplus (y \oplus z)$. Notice that the (syntactic) binding conventions of an operator have nothing to do with the (semantic) notion of *associativity*, except for the fact that associative operators need no separate binding conventions.

Function application is denoted by the infix operator $\cdot$ ("dot"). For example, we write $f \cdot x$ and $g \cdot x \cdot y$ instead of the more classical $f(x)$ and $g(x,y)$. Operator $\cdot$ is left-binding and it binds stronger than all other operators. Sometimes, we use subscription for function application, which binds even stronger than $\cdot$; for example, $f \cdot x_y$ means $f \cdot (x \cdot y)$.

The most important property of functions is $x = y \Rightarrow f \cdot x = f \cdot y$; its explicit formulation is attributed to Leibniz. In calculations, we use the hint "Leibniz" to indicate use of this property. In the more syntactic realm of formula manipulation this is also called *substitution of equals for equals*.

A *predicate on set* V is a boolean-valued function on V . Actually, we do not formally distinguish predicates on a set from subsets of that set: we identify each subset with its characterising predicate. So, we write $U \cdot x$ instead of $x \in U$, we have $P = \{ x \mid P \cdot x \}$, and $\{ x \}$ denotes the point-predicate that is true in x only.

For predicates we use the calculus developed by E.W. Dijkstra and W.H.J. Feijen [Dij3]. In order to make this monograph more self-contained, we summarise the rules for quantified expressions here.

With any symmetric and associative binary operator $\oplus$ on a set V , a, so-called, *quantifier* is associated, denoted here by $\bigoplus$. With it we can form quantified expressions of the form $(\bigoplus x : P \cdot x : F \cdot x)$ in which name x occurs as a *dummy* (bound variable), and in which P is a predicate on a set U , say, and F is a function of type $U \rightarrow V$. In practice, $P \cdot x$ and $F \cdot x$ are often (represented by) expressions in which x may occur as a free variable. Predicate P is a subset of U called the *range* of the quantification. For the sake of brevity, $P \cdot x$ is sometimes omitted, giving $(\bigoplus x :: F \cdot x)$. Expression $F \cdot x$ is called the *term* of the quantification. The most important rules for manipulating quantified expressions are:

empty-range rule: $(\bigoplus x : \mathrm{false} : F \cdot x) = e$,
provided that $\oplus$ has identity e .

one-point rule: $(\bigoplus x : x = y : F \cdot x) = F \cdot y$, for all $y : U \cdot y$.

range split: $(\bigoplus x : P{\cdot}x \vee Q{\cdot}x : F{\cdot}x) = (\bigoplus x : P{\cdot}x : F{\cdot}x) \oplus (\bigoplus x : Q{\cdot}x : F{\cdot}x)$, provided that $\oplus$ is idempotent or that P and Q are disjoint.

dummy substitution: $(\bigoplus x : P{\cdot}x : F{\cdot}x) = (\bigoplus y : P{\cdot}(f{\cdot}y) : F{\cdot}(f{\cdot}y))$, where function f is either bijective or surjective; in the latter case $\oplus$ must be idempotent.

dummy shuffling: $(\bigoplus x : P{\cdot}x : (\bigoplus y : Q{\cdot}y : F{\cdot}x{\cdot}y)) =$
$(\bigoplus y : Q{\cdot}y : (\bigoplus x : P{\cdot}x : F{\cdot}x{\cdot}y))$.

Because of the possibility of dummy shuffling, nested quantifications can also be written, without causing confusion, as $(\bigoplus x,y : P{\cdot}x \wedge Q{\cdot}y : F{\cdot}x{\cdot}y)$.

The most frequently used quantifiers are:

A	corresponding to $\wedge$	(idempotent, with identity true)
E	corresponding to $\vee$	(idempotent, with identity false)
S	corresponding to +	(not idempotent, with identity 0)
MIN	corresponding to **min**	(idempotent, with identity ∞)
MAX	corresponding to **max**	(idempotent, with identity $-\infty$) .

For boolean quantifications we have the possibility of, so-called, *range* and *term trading*, and of *instantiation*; the rule of instantiation can be derived by means of range split and the one-point rule:

trading: $(\mathbf{A}x : P{\cdot}x \wedge Q{\cdot}x : R{\cdot}x) \equiv (\mathbf{A}x : P{\cdot}x : Q{\cdot}x \Rightarrow R{\cdot}x)$
$(\mathbf{E}x : P{\cdot}x \wedge Q{\cdot}x : R{\cdot}x) \equiv (\mathbf{E}x : P{\cdot}x : Q{\cdot}x \wedge R{\cdot}x)$.

instantiation: $(\mathbf{A}x : P{\cdot}x : Q{\cdot}x) \Rightarrow Q{\cdot}y$, for all $y : P{\cdot}y$
$(\mathbf{E}x : P{\cdot}x : Q{\cdot}x) \Leftarrow Q{\cdot}y$, for all $y : P{\cdot}y$.

Furthermore, **MIN** and **MAX** have the following properties, which may be considered as their definitions:

definition of **MIN**: $F{\cdot}y = (\mathbf{MIN}\,x : P{\cdot}x : F{\cdot}x) \equiv P{\cdot}y \wedge (\mathbf{A}x : P{\cdot}x : F{\cdot}y \leq F{\cdot}x)$, for all $y : U{\cdot}y$.

definition of **MAX**; $F{\cdot}y = (\mathbf{MAX}\, x : P{\cdot}x : F{\cdot}x) \equiv P{\cdot}y \wedge (\mathbf{A}\, x : P{\cdot}x : F{\cdot}x \leqslant F{\cdot}y)$,
for all $y : U{\cdot}y$.

Finally, if another operator $\otimes$, say, distributes over $\oplus$, then it also distributes over quantified expressions with *nonempty* range; i.e. for P , $P \neq \emptyset$, we have:

distribution;

$$(\oplus\, x : P{\cdot}x : F{\cdot}x) \otimes y = (\oplus\, x : P{\cdot}x : F{\cdot}x \otimes y)$$
$$y \otimes (\oplus\, x : P{\cdot}x : F{\cdot}x) = (\oplus\, x : P{\cdot}x : y \otimes F{\cdot}x) \quad .$$

The two rules of distribution are also valid for empty P , provided that $e \otimes y = e$ and $y \otimes e = e$ respectively. For example, because $\mathrm{true} \vee y \equiv \mathrm{true}$ and $y \vee \mathrm{true} \equiv \mathrm{true}$, $\vee$ distributes over $\mathbf{A}$ in all cases; similarly, $\wedge$ distributes over $\mathbf{E}$.

Quantified expressions inherit many other properties of the operators on which they are based. For example, the rules of De Morgan also apply to boolean quantifications.

We illustrate the use of the above conventions with a small example of a derivation. In practice, range splits in which a single point of the range is split off followed by an application of the one-point rule are combined into a single step. Because this derivation exhibits a pattern that occurs quite often, we sometimes even combine all four steps of this derivation into one step; for j satisfying $0 \leqslant j$ we have:

$$\begin{array}{ll}
 & (\mathbf{S}\, i : 0 \leqslant i < j{+}1 : F{\cdot}i) \\
= & \quad \{ \text{ range split: } 0 = i \vee 1 \leqslant i < j{+}1 \ \ (\text{using } 0 \leqslant j) \ \} \\
 & (\mathbf{S}\, i : 0 = i : F{\cdot}i) + (\mathbf{S}\, i : 1 \leqslant i < j{+}1 : F{\cdot}i) \\
= & \quad \{ \text{ one-point rule } \} \\
 & F{\cdot}0 + (\mathbf{S}\, i : 1 \leqslant i < j{+}1 : F{\cdot}i) \\
= & \quad \{ \text{ dummy substitution } i \leftarrow i{+}1 \ (\text{i.e: } f{\cdot}i = i{+}1) \ \} \\
 & F{\cdot}0 + (\mathbf{S}\, i : 1 \leqslant i{+}1 < j{+}1 : F{\cdot}(i{+}1)) \\
= & \quad \{ \text{ algebra } \} \\
 & F{\cdot}0 + (\mathbf{S}\, i : 0 \leqslant i < j : F{\cdot}(i{+}1)) \quad .
\end{array}$$

1 The basic formalism

1.0 Introduction

In this chapter we define a simple, abstract functional-program notation called the *basic formalism*. The basic formalism constitutes the essence of functional programming, as we view it, and nothing else. The values of the expressions in the formalism may be *interpreted* as functions, but otherwise these values are uninterpreted objects. In this respect, the basic formalism resembles *λ-calculus*, but, in contrast to λ-calculus, the notation has been designed for programming. Particularly, the notation differs from λ-calculus by the use of, so-called, *where-clauses* instead of λ-abstractions; the idea to use where-clauses has been adopted from notations such as SASL [Tur0].

Although simple and abstract, the basic formalism is complete: if so desired, all features of the program notation, as it is developed in this monograph, can be defined in the basic formalism. Thus, the basic formalism provides a simple setting for the discussion of a number of general conventions and theorems. These conventions and theorems, then, are applicable to the full program notation too.

The basic formalism consists of a set of, so-called, *expressions*, a set of, so-called, *values*, and a, so-called, *value function* that maps expressions to values. So, a value is assigned to every expression. The expressions form the syntactic part of the formalism, whereas the values and the value function form the semantics of the formalism. In this monograph we define the semantics axiomatically, by means of postulates specifying properties of the set of values and of the value function. These postulates capture all we need for the sake of programming; yet, they do not define the values and the value function completely. This does not mean, however, that we propose a *nondeterministic* program notation. By means of the value function, the relation between an expression and its value is supposed to be completely fixed; the *indeterminacy*, as we propose to call it, of the program notation is a property of the postulates specifying the relation between expressions and their values, not of that relation itself. We have chosen this modus operandi in order to avoid over-specification: we do not wish to define those properties of the value function that we consider irrelevant for our purpose, namely the derivation of programs from specifications.

We think that this approach yields the simplest possible formalism suitable for programming. One marked difference between our approach and other presentations of the subject is that we deliberately omit all notions regarding the possibility to interpret expressions as executable programs. This does not mean that this *operational interpretation* has not played a role in the design of the formalism. On the contrary! The way in which the rules of the game have been chosen can only be justified by an appeal to the requirement that the formalism allows a -- sufficiently efficient -- operational interpretation. Yet, it is possible to explain and use the formalism, free from connotations, as a piece of mathematics in its own right. Particularly, within the formalism there is no such notion as *termination*, and we need not concern ourselves with the termination of -- the computations evoked by execution of -- our programs. We discuss these operational aspects of the notation in more detail in chapter 3.

In order that the formalism be useful for practical purposes, we must prove that the set of rules defining it is *consistent*, i.e. non-contradictory. In this monograph we do not present such proof. This proof is, however, not difficult to construct, if we take into account the following two observations. First, the basic formalism can be translated easily into λ-calculus, in such a way that the, so-called, *term model* [Hin] guarantees its consistency. Second, as stated above, all features of the full notation can be defined in the basic formalism. Hence, the consistency of the program notation follows from the consistency of the basic formalism.

In the next sections we present the basic formalism without explaining why we have chosen it to be the way it is. Apart from its usability for programming, the main design criterion has been our explicit desire not to distinguish formally between *functions* and *other values*. Therefore, in the basic formalism the notion of a function does not occur; it is only a matter of *interpretation* that we consider values as functions. We discuss this in section 1.9.

1.1 Values and expressions

The formalism consists of a set of *expressions*, a set of *values*, and a *value function*, mapping expressions to values. The set of expressions is called Exp , the set of values is called Ω . The value function remains anonymous: all of its properties relevant to our purpose can be expressed by means of equalities between the values of expressions. Therefore, for expressions E and F , we use $E = F$ to denote semantic equality, not syntactic equality, of E and F . In the, rather exceptional, case where we wish to discuss syntactic equality of expressions, we announce this explicitly. Notice that this is common practice in mathematics. For example, the assertion $2+3 = 5$ means that expressions $2+3$ and 5 denote the same values; it does not mean that they are the same expressions. In what follows we do not explicitly mention the value function anymore; instead, we simply speak of the value of an expression.

The expressions are symbolic representations of the (abstract) values in Ω : the values in Ω are the objects of our interest, whereas the expressions *only* serve to provide us with representations (of these values) that can be manipulated in derivations and that can be evaluated by machines. This implies that with each value in Ω and with each operator on Ω there is a corresponding syntactic form representing it. Algebraically speaking this means that the value function is a *homomorphism*: if operator $+$, of type $\Omega \times \Omega \to \Omega$, is rendered syntactically by symbol $\oplus$, then we have, for expressions E and F with values x and y , that the value of $E \oplus F$ is $x+y$. This being so, we can, to a large extent, identify expressions with their values. Actually, it would be nice if we could ignore syntactic considerations altogether and play the game in a purely algebraic way. The use of substitution, in the rules of *folding and unfolding*, however, prohibits this.

We start with a discussion of some properties of Ω . Within the basic formalism the elements of Ω are uninterpreted objects, simply called *values*. Ω is the universe containing all values we will be studying.

convention 1.1.0: From here onwards all variables in our formulae range over Ω , unless stated otherwise.

□

In order to exclude the, trivial, one-point model, we require that Ω is not a singleton set. Later we show that $\Omega \neq \emptyset$.

postulate 1.1.1 (Ω is not a singleton): $(\mathbf{A}x :: (\mathbf{E}y :: x \neq y))$
□

Next, we assume the existence of a binary operator, denoted by $\cdot$ ("dot"), of type: $\Omega \times \Omega \rightarrow \Omega$. This operator is written in infix notation, and, in order to save parentheses, we adopt the convention that it is left-binding; i.e. $x \cdot y \cdot z$ must be parsed as $(x \cdot y) \cdot z$. Moreover, it binds stronger than any other operator.

note: Here we use the same symbol that we use for function application in our metanotation. This causes no confusion, provided that it is always clear what the types of the operands are. On the other hand, this convention turns out to be extremely convenient: later, when we interpret values in Ω as functions, operator $\cdot$ happens to represent function application.
□

In order to be able to reason about values in Ω , we need expressions to represent them. For this purpose we define the set Exp , of expressions, temporarily as follows.

definition 1.1.2 (syntax of expressions):

(0) Exp → Const

(1) Exp → Name

(2) Exp → '(' Exp '·' Exp ')'

□

Syntactic category Const will be defined later; its elements are called *constants*. The idea is that constants represent (well-defined) elements of Ω , namely solutions of *equations* of a particular kind. Syntactic category Name is left unspecified here; its elements are *names* -- in the, more or less, usual meaning of the word -- . In expressions names are *only* used as *dummies*: they are bound by universal quantification or by, so-called, *where-clauses*, by means of which constants are defined. In either case they represent values in Ω . We assume that Name is infinite; thus, we have an infinite supply of *fresh names*, i.e. names not occurring (yet) in our expressions.

Rule (2) in the above definition provides a syntactic representation of operator $\cdot$ on Ω . This rule prescribes that all composite expressions should be parenthesized. In practice, however, we omit parentheses whenever this causes no ambiguity. Obviously, for symbol $\cdot$ we adopt the same binding conventions as for the corresponding operator; as explained above, we ignore the distinction between operators on Ω and their syntactic representations as much as possible.

1.2 Intermezzo on combinatory logic

The development of the formalism can now be continued in various ways. One way, which we shall not pursue further, is to postulate the existence, in Ω , of a fixed set of constants with a number of explicitly formulated properties. We may, for instance, introduce constants I , K , and S , and postulate that they have the following properties:

(0) $(\mathbf{A}\, x :: I\cdot x = x)$

(1) $(\mathbf{A}\, x,y :: K\cdot x\cdot y = x)$

(2) $(\mathbf{A}\, x,y,z :: S\cdot x\cdot y\cdot z = (x\cdot z)\cdot(y\cdot z))$.

With these, so-called, *combinators* the same games can be played as with our formalism. It is, for instance, possible to prove that (0) , as a postulate, is superfluous: I can be defined in terms of K and S , for example by $I = S\cdot K\cdot K$. Then, (0) follows from (1) and (2) . In the same vein, more interesting combinators can be defined in terms of K and S , such as combinator Y satisfying:

(3) $(\mathbf{A}\, x :: Y\cdot x = x\cdot(Y\cdot x))$.

This shows that every value in Ω , when considered as a function, has *a fixed point*. One possible definition of Y is:

$$Y = S\cdot W\cdot W \text{ , with}$$
$$W = S\cdot(S\cdot(K\cdot S)\cdot K)\cdot(K\cdot(S\cdot I\cdot I)) \text{ .}$$

The formalism thus obtained is called *combinatory logic* [Hin]. Its advantage is that rule (1) in definition 1.1.2 can be abolished: the whole game can be played without the use of names. Thus, the mathematically awkward notion of substitution is avoided. Although interesting from a mathematical point of view, and although useful for the implementation of functional-program notations [Tur1][Pey], combinatory logic is ill-suited for programming.

1.3 The dot postulate

Let E be an expression in which no other names occur than x, y, z. We now consider the following equation:

(0) $x\colon (\mathbf{A}\, y,z :: x\cdot y\cdot z = E)$.

Notice that the names occuring in expressions are used as dummies: systematic replacement of one of the names by a fresh one transforms the equation into the *same* equation. In equation (0) , x is the *unknown*, y and z are the *parameters of* x , and E is called the *defining expression of* x . Since x may occur in its defining expression, equations of this type are also called *recursion equations* [Tur2].

Equation (0) is rather special in that its unknown has 2 parameters: the unknown may have any (natural) number of parameters. The restriction imposed here on E -- later this restriction is relaxed again -- is that the only names occurring in it are the unknown and its parameters. We call equations formed according to these rules *admissible equations*. The following example shows that, even without the use of constants, it is possible to construct admissible equations.

example 1.3.0: Here are a few admissible equations:

(0 parameters):	$x\colon x = x$
(0 parameters):	$x\colon x = x\cdot x$
(1 parameter) :	$x\colon (\mathbf{A}\, y :: x\cdot y = y)$
(2 parameters):	$x\colon (\mathbf{A}\, y,z :: x\cdot y\cdot z = y)$
(2 parameters):	$x\colon (\mathbf{A}\, y,z :: x\cdot y\cdot z = z\cdot x\cdot y)$

□

The following postulate is the characteristic postulate of the basic formalism, in the sense that it has far-reaching consequences.

postulate 1.3.1 (dot postulate): Every admissible equation has a (i.e: at least one) solution in Ω .

□

property 1.3.2: $\Omega \neq \emptyset$, because there are admissible equations.

property 1.3.3: Postulates (0), (1), (2) in section 1.2 can be considered as admissible equations, with unknowns I, K, and S respectively. Hence, Ω contains values I, K, and S satisfying these postulates, and our formalism contains combinatory logic.

property 1.3.4: It would have been sufficient to state the dot postulate for *non-recursive* equations -- i.e. equations in which the unknown does not occur in its defining expression -- only. By means of a combinator Y , satisfying (3) in section 1.2, we can prove that each recursive equation has a solution too.

property 1.3.5: By means of combinators I and K only, it can be proved that Ω is either a singleton set or infinite. Using postulate 1.1.1 we conclude that Ω is infinite.

□

aside 1.3.6: The kind of equations introduced here is typical for functional-progam notations. By admission of other -- usually larger -- classes of equations other formalisms can be obtained, such as *logic-program* notations. A larger class of admissible equations makes programming easier, but can be more difficult to implement efficiently: each program notation is a compromise between ease of programming and implementability [Hoa0]. The advantage of functional notations over logic notations is that the former can be implemented much more efficiently than the latter.

□

1.4 Where-clauses

Due to the dot postulate, every admissible equation has a solution in Ω . We now extend the formalism with a notation for these solutions; they form the constants mentioned, but left unspecified, in section 1.1. Constants are denoted by *names* (as are parameters). Such names are bound to the values they represent by means of, so-called, *where-clauses*. A where-clause contains a name and -- a concise encoding of -- an admissible equation; its meaning is that, within the expression to which the where-clause pertains, the name represents a solution of the equation. On account of the dot postulate, such a solution exists.

It is, of course, possible that the equation has many solutions. In that case, we leave unspecified which solution is intended, but we do postulate that all occurrences of the constant thus defined denote *one and the same solution* of the equation. The latter requirement is necessary to keep the formalism deterministic. As a result, the only thing we (care to) know about the constant is that it is a solution of its defining equation, and this is the only knowledge we shall ever use. The proof and manipulation rules formulated in the next section are based on this attitude.

A where-clause serves two purposes: it provides a syntactic representation of a (particular) solution of an admissible equation, and it provides a name for that solution. The reason to use names to denote constants is threefold. First, since names are now used for constants and for parameters, the roles of constants and parameters can be interchanged, if so desired. Second, the use of a name enhances modularisation, by providing a clear separation of the *definition* of a constant from its *use*; even if the name is used only once in the program, the increase in clarity usually outweighs the price of its introduction. Third, the use of names allows recursive definitions.

The syntax of expressions can be defined as follows. Notice that, because constants are represented by names, the syntactic category Const can now be abolished. The following definition replaces definition 1.1.1.

definition 1.4.0 (syntax of expressions):

Exp → Name
Exp → '(' Exp '·' Exp ')'
Exp → '(' Exp '⟦' Def '⟧' ')'
Def → Name { '·' Name } '=' Exp .

The elements of syntactic category Def are called *definitions*. Constructs of the form { '·' Name } are called *parameter lists*. All names occurring to the left of the = sign in a definition must be different.

□

suggested pronunciation: Pronounce $[\![$ as "where" and $]\!]$ as "end(where)". □

Syntactic category Def has been introduced so that we can extend it later. We introduce some nomenclature, using the following example expression, where E and F are expressions and x, y, z are names:

(0) $F [\![x \cdot y \cdot z = E]\!]$.

Construct $[\![x \cdot y \cdot z = E]\!]$ is called a *where-clause*. It *defines* constant x ; i.e. within F , each occurrence of x denotes that constant. Thus, F constitutes the *scope* of x . Within expression (0) , when treated as a whole, x is a dummy: "outside" (0) , occurrences of x have no meaning, at least not on account of the where-clause in (0) .

Definition $x \cdot y \cdot z = E$ is x's *defining equation* or *definition*, for short; it is an abbreviation of the equation x: $(\mathsf{A}\, y,z :: x \cdot y \cdot z = E)$. Notice that such abbreviated notation is possible, without causing confusion, due to the restricted form of admissible equations. Furthermore, notice that, here, we use x in two ways: on the one hand, it denotes the unknown of the equation; on the other hand, it denotes -- in F -- one particular solution of that equation.

As explained earlier in section 1.3, in expression E both x and its parameters, y and z , may occur. Occurrences of a name in its (own) defining expression (cf. section 1.3) are called *recursive occurrences*. Moreover, expression (0) may occur as subexpression in a larger expression. In that case, both E and F may contain other names representing constants or parameters of constants defined in where-clauses in the larger expression: where-clauses may, for instance, be nested.

Summarising, we conclude that there are three ways in which a name can occur in an expression:

- as a constant,
- as a parameter,
- as a recursive occurrence.

The latter two cases pertain only to expressions that occur in the right-hand side of a defining equation, whereas occurrence as a constant requires that the expression is (part of) an expression containing a where-clause defining that name.

1.5 Free names, programs, and substitution

In the above, we used the notion "occur in (an expression)" in a rather loose sense: actually, we meant "*occur as a free name*". Throughout this monograph we use "occur in" in this meaning. A formal definition, by recursion on the definition of expressions (cf. definition 1.4.0) is:

definition 1.5.1 (names occurring in an expression):
For expressions E, F , *different* names x, y , and parameter list pp :

"x occurs in x"
¬ "x occurs in y"
"x occurs in E·F" ≡ "x occurs in E" ∨ "x occurs in F"
¬ "x occurs in F|[x pp = E]|"
"x occurs in F|[y pp = E]|" ≡
"x occurs in F" ∨ ("x occurs in E" ∧ ¬ "x occurs in pp")

□

definition 1.5.2 (program): A *program* is an expression in which no (free) names occur.

□

Because names are only used as dummies and constants, we cannot assign a value to expressions in which free names occur. Hence, the above definition of programs. When taken in isolation, however, the subexpressions of an expression may contain free names. This causes no problems, because such subexpressions must always be understood in the *context* of the whole expression they are part of. For reasons of manipulative efficiency it often happens that we study subexpressions in isolation: we do not wish to copy, over and over again, those parts of the expression that remain constant in the derivation. Particularly, where-clauses are almost never subjected to formal manipulation. They only provide definitions of the constants occurring in our expressions. Therefore, we omit them when we manipulate these expressions.

example 1.5.3: $I \,|[I \cdot y = y]|$ and $C \,|[C \cdot x = I \,|[I \cdot y = y]| \,]|$ are programs. We prove that their values are different, by manipulation of C and I :

$C \neq I$

$\Leftarrow$ { Leibniz, heading for applicability of C's and I's definitions }

$(E\, y :: C \cdot y \neq I \cdot y)$

$=$ { definitions of C and I }

$(E\, y :: I \neq y)$

$=$ { postulate 1.1.1 with $x \leftarrow I$ }

true .

□

An often used kind of formula manipulation is *substitution*; it is a textual operation mapping one expression to another by replacement of all free occurrences of a name by (copies of) an expression. For the sake of completeness, we give a definition of substitution for the basic formalism, but we shall not use it explicitly; the reader is assumed to know the notion from other formalisms and he is assumed to be able to identify, in expressions, the free occurrences of a name.

definition 1.5.4 (substitution): For name x and expressions E and G, $E(x \leftarrow G)$ denotes the expression obtained from E by replacement of all free occurrences of x by G. Formally, for expressions E, F, G, different names x, y, and parameter list pp :

$$
\begin{array}{lll}
 & x(x \leftarrow G) & = G \\
 & y(x \leftarrow G) & = y \\
 & (E \cdot F)(x \leftarrow G) & = E(x \leftarrow G) \cdot F(x \leftarrow G) \\
 & F|[x\ pp = E]|(x \leftarrow G) & = F|[x\ pp = E]| \\
(0) & F|[y\ pp = E]|(x \leftarrow G) & = F(x \leftarrow G)|[y\ pp = E]| \quad , \text{ if } pp \text{ contains } x \\
(1) & F|[y\ pp = E]|(x \leftarrow G) & = F(x \leftarrow G)|[y\ pp = E(x \leftarrow G)]| \quad , \text{ if } x \text{ not in } pp \ .
\end{array}
$$

Rule (0) is only correct for y not occurring in G. If y occurs in G, systematic replacement of y by a fresh name z, say, does the job; i.e, replace the expression $F|[y\ pp = E]|$ by $F(y \leftarrow z)|[z\ pp = E(y \leftarrow z)]|$. Because z is fresh, it is does not occur in G ; so, rule (0) is applicable to the new expression.

The same holds for rule (1) ; moreover, rule (1) is only correct if none of y's parameters occur in G . By a similar systematic replacement of these parameters and their occurrences in E , by fresh names, the where-clause can be transformed into one in which no parameter occurs in G .

□

note 1.5.5: The equal signs in this definition denote *syntactic* equality; for example: $x(x \leftarrow G)$ and G are, by definition, the same expression. Furthermore, that the result of a substitution is indeed an expression requires proof, but this is easy.

□

1.6 Multiple definitions

We extend the basic formalism a little further. For expressions E and F , and names u, v, w, x, y , we consider the following example of a *simultaneous equation*, with unknowns x and y :

(0) $\quad x,y\!: (\mathbf{A}\, u,v :: x{\cdot}u{\cdot}v = E) \wedge (\mathbf{A}\, w :: y{\cdot}w = F)$.

Such equations are particularly interesting when x occurs in F and y occurs in E ; this is known as *mutual recursion*. We now decide that such simultaneous equations, with any number of unknowns, each with its own number of parameters, are admissible too; hence, the dot postulate also pertains to these equations. In definitions, we simply write $x{\cdot}u{\cdot}v = E \;\&\; y{\cdot}w = F$, where the new symbol & ("and") is used to combine two definitions into one, so-called, *multiple definition*.

This extension of the formalism is captured by the following extension of the syntax rules of the notation:

definition 1.6.0 (multiple definitions; extension of definition 1.4.0):

Def → Def '&' Def

□

note 1.6.1: The symbol & corresponds with the, symmetric and associative, $\wedge$ occurring in the equations corresponding to the definition; hence, we consider the syntactic ambiguity introduced by definition 1.6.0 as harmless. Moreover, the order in which definitions are combined into one multiple definition is irrelevant. In this sense & may, although it is not an operator, be considered as being symmetric and associative.

The introduction of multiple definitions is not a true generalisation. By means of *tupling*, introduced in chapter 2, these definitions can be transformed into equivalent, simple definitions.

□

1.7 Proof and manipulation rules

In this section we discuss a number of formal rules for the manipulation of expressions. Roughly, these rules come in two kinds. First, there is a rule for proving properties of expressions; second, there are rules for transforming expressions into other expressions. Both kinds of rules can be justified by interpretation of the expressions involved, using the definitions given in the previous sections.

1.7.0 introduction and elimination of where-clauses

The following rule does not depend on the special form of admissible equations, as defined in sections 1.3 and 1.6. Therefore, we use a generalised form of where-clauses: with Q a predicate on Ω, such that $(\mathsf{E}\,x :: Q{\cdot}x)$, we use the where-clause $|[x : Q{\cdot}x]|$ to indicate that x is defined to be a value satisfying $Q{\cdot}x$. Similarly, we use $|[x,y : Q{\cdot}x{\cdot}y]|$. Later, we also use such forms in the development of programs to *specify* values for which subprograms must still be developed. The rule given here allows us to prove properties of an expression with a where-clause in terms of its subexpressions. The rule treats *all* solutions of the equation represented by the where-clause on equal footing; thus, the rule reflects that we leave unspecified which solution is represented by the name defined in the where-clause.

rule 1.7.0.0 (proof rule for where-clauses): For predicate Q, satisfying $(\mathbf{E}\, x :: Q{\cdot}x)$, predicate R on Ω, expression F, and name x:

$$R{\cdot}(F[\![x : Q{\cdot}x]\!]) \Leftarrow (\mathbf{A}\, x : Q{\cdot}x : R{\cdot}F) \quad .$$

For the case of a multiple definition in which k constants are defined, Q becomes a predicate with k arguments, and the dummy x in the above formulae must be replaced by k dummies representing these constants. For the case $k = 2$, we thus obtain:

$$R{\cdot}(F[\![x,y : Q{\cdot}x{\cdot}y]\!]) \Leftarrow (\mathbf{A}\, x,y : Q{\cdot}x{\cdot}y : R{\cdot}F) \quad .$$

□

example 1.7.0.1: For expressions E and F, name x such that x does not occur in E, and value X, we derive:

$F[\![x = E]\!] = X$

$\Leftarrow$ { rule 1.7.0.0 ($[\![x = E]\!]$ means $[\![x : x = E]\!]$), with $R{\cdot}x \leftarrow x = X$ }

$(\mathbf{A}\, x : x = E : F = X)$

$=$ { x does not occur in E; one-point rule }

$F(x \leftarrow E) = X$.

Choosing $F(x \leftarrow E)$ for X, we conclude that, when x does not occur in E, $F[\![x = E]\!] = F(x \leftarrow E)$.

□

In this example we have derived one of the following properties. The other ones can be derived in a similar way.

property 1.7.0.2 (where-clause elimination rules):

for x not in E:	$F[\![x = E]\!]$	$=$	$F(x \leftarrow E)$
for x not in F:	$F[\![x = E]\!]$	$=$	F
for x not in F:	$F[\![x\ pp = E]\!]$	$=$	F

□

1.7.1 folding and unfolding

Substitution is an operation by which *all* occurrences of a name are replaced by an expression. In program derivations, particularly in program transformations, we wish to be able to replace a *single* occurrence of a name by an expression. The inverse operation, replacing a subexpression by a name, is of interest too. Both kinds of manipulation are made possible by the following rule. We content ourselves with an informal formulation, for two reasons. First, in practical situations, everybody with a modest experience in formula manipulation is able to apply this rule without error. Second, the notions of a single (free) occurrence of a name and replacement thereof are hard to formalise. The rule is formulated here for names with 2 parameters only.

rule 1.7.1.0 (rule of folding and unfolding): For expressions A, B, E, F, G and names x, y, z : if replacement, in F , of subexpression $x{\cdot}A{\cdot}B$ by $E(y,z \leftarrow A,B)$ yields G , then we have:

$$F\,[\![x{\cdot}y{\cdot}z = E]\!] \;=\; G\,[\![x{\cdot}y{\cdot}z = E]\!] \;.$$

If we use this rule to obtain $G\,[\![x{\cdot}y{\cdot}z = E]\!]$ from $F\,[\![x{\cdot}y{\cdot}z = E]\!]$, then we call this *unfolding x* ; if used to obtain the former expression from the latter, then this is called *folding x* . The rule may also be applied when the where-clause contains multiple definitions, i.e. when it is of the form $[\![x{\cdot}y{\cdot}z = E \;\&\; D]\!]$, for any definition D .

□

example 1.7.1.1: For expressions E, F and names x, y , such that y does not occur in F , we derive:

$$\begin{array}{ll}
 & x{\cdot}F\,[\![x{\cdot}y = E]\!] \\
= & \quad \{\text{ unfolding } x \;\} \\
 & E(y \leftarrow F)\,[\![x{\cdot}y = E]\!] \\
= & \quad \{\; y \text{ not in } F\text{: property 1.7.0.2 } \} \\
 & E\,[\![y = F]\!]\,[\![x{\cdot}y = E]\!] \;.
\end{array}$$

In expression $x{\cdot}F\,[\![x{\cdot}y = E]\!]$, y occurs as a parameter, whereas in subexpression $E|[y=F]|$ it occurs as a constant. Apparently, parameters and constants are not so different as they may seem at first sight.

□

example 1.7.1.2: For expression E and name x, we derive:

E

= { let y be a fresh name: property 1.7.0.2 }

$E|[y{\cdot}x=E]|$

= { folding y }

$y{\cdot}x|[y{\cdot}x=E]|$

= { shunting rule, see below }

$(y|[y{\cdot}x=E]|)\cdot(x|[y{\cdot}x=E]|)$

= { y not in x: property 1.7.0.2 }

$(y|[y{\cdot}x=E]|)\cdot x$.

□

corollary 1.7.1.3: For every expression E and name x, an expression F, in which x does not occur, exists such that $E=F{\cdot}x$.

□

1.7.2. shunting

The following rule expresses that where-clauses may be distributed over dots.

rule 1.7.2.0 (shunting rule): For expressions E and F, and definition D :

$$E{\cdot}F|[D]| = (E|[D]|)\cdot(F|[D]|)$$

□

The shunting rule is used to separate/apply expressions denoting functions from/to their arguments, as in example 1.7.1.2; applied in combination with property 1.7.0.2, we have $x{\cdot}A{\cdot}B|[x{\cdot}y{\cdot}z=E]| = (x|[x{\cdot}y{\cdot}z=E]|){\cdot}A{\cdot}B$, provided that x does not occur in either A or B . Although we usually prefer to write expressions in the form $x \cdot A \cdot B \,[\![x \cdot y \cdot z = E]\!]$, we sometimes wish to consider $x \,[\![x \cdot y \cdot z = E]\!]$ in isolation.

1.8 Specifications, programming, and modularisation

In this section we define specifications and we discuss the relation between specifications and programs. Furthermore, we briefly discuss the topic of modularisation.

definition 1.8.0 (specification): A specification is a predicate on Ω .
□

In practical situations a specification is, of course, a mathematical expression denoting a predicate on Ω . If we would play the game really formally, we should also define a *specification notation* in a formal way. In this monograph, we do not do so, but we use, more or less common, mathematical formulae instead.

In very general terms, *programming* can now be defined as the activity of constructing a program -- cf. definition 1.5.2 -- whose value satisfies an, a priori given, specification. Usually, the program constructed must also satisfy certain *efficiency requirements*. For the time being, we ignore these. We now interpret this definition in a few, slightly different, ways.

Let R be a specification. A value satisfying it can be denoted by the following *proto program* -- here we use the generalised form of where-clauses introduced insection 1.7.0 -- :

(0) $x \,[\![x : R \cdot x]\!]$.

We call this a proto program because $x : R \cdot x$ is not a definition in the program notation. If, however, $x : R \cdot x$ is an admissible equation, then (0) can be encoded straightforwardly into a correct program, and we are done. If not, we can try to *transform* (0) , by means of formula manipulation, in as many steps as needed to obtain a program. The same approach can be applied if we have a program, but if we are not yet satisfied with its efficiency. This is called the *transformational* approach to programming. We would like to stress here that the starting point of such a transformation process need not be a program: it may be any specification.

When, in a sequence of transformations starting with (0), only predicate R is manipulated, then the constant obligation to copy, in each step, the symbols surrounding R becomes a little annoying. More importantly, by confining our manipulations to the predicate, we obtain the freedom to *strengthen* it: if $(\mathrm{A}x :: R{\cdot}x \Leftarrow Q{\cdot}x)$, for some predicate Q, then (1) satisfies R -- this follows directly from the proof rule for where-clauses --, with:

(1) $\quad x [\![x : Q{\cdot}x]\!]$.

We must only convince ourselves that $x : Q{\cdot}x$ has a solution; if this equation is an admissible one then this obligation is void, because the dot postulate does the job. So, if (1) is a program we are done.

This approach gives rise to a style of programming that takes place mainly in the domain of predicate calculus. It can also be considered as *transformational*: the specification is transformed, viz. strengthened, into one that is an admissible definition.

The value of the program to be constructed may also be specified by a mathematical expression denoting that value. Actually, formula (0) is a special case of this. Again, we may try to transform this expression into an equivalent expression in the program notation. We can, however, also bring this problem into the domain of predicate calculus, by defining predicate R by: $(\mathrm{A}x :: R{\cdot}x \equiv (x = E))$, where E denotes the expression specifying the program.

In the above, we have shown that programming can be carried out both in the domain of expressions and in the domain of predicate calculus. Moreover, transitions between the two domains can be performed quite easily. In practical situations, we choose the domain that suits our purpose best.

* * *

We now consider expressions of the following form:

(2) $\quad F [\![x = E]\!]$.

For the sake of simplicity, we assume here that x has no parameters. The way in which expressions of this form are constructed can be characterised as follows. Starting with a specification R , we derive a, tentative, expression G such that $R{\cdot}G$. Now assume that G contains, one or more, instances of a subexpression that is not yet an expression in the program notation. We then may decide to give this subexpression a name x and replace all of its instances by x . This yields expression F . Moreover, from the information obtained in the derivation of G, we construct a specification Q for x , in such a way that $(\mathbf{A}x : Q{\cdot}x : R{\cdot}F)$. Using the proof rule for where-clauses, we conclude $R{\cdot}(F[\![x : Q{\cdot}x]\!])$. Finally, from Q we derive the definition $x = E$. Hence, the correctness of expression F only depends on x's specification, not on its definition. Thus, x's specification is the interface between subexpressions F and E . For instance, if we replace E by a new expression E' , then the only new proof obligation is $(\mathbf{A}x : x = E' : Q{\cdot}x)$; this replacement generates no new proof obligations for F . Thus, by constructing specifications for names defined in where-clauses, we can use where-clauses to construct programs in a *modular* way.

1.9 Functions

The elements of Ω are uninterpreted, abstract objects. We show that they can be interpreted as *functions* of type $\Omega \to \Omega$. Let f be a fixed element of Ω . We define function $F : \Omega \to \Omega$, as follows -- where we use, for the sake of clarity, classical notation for function application -- :

(0) $(\mathbf{A}x :: F(x) = f{\cdot}x)$.

Thus, f is a representation of F , and Ω is a representation of a *subset* of $\Omega \to \Omega$. Of course, not every function in $\Omega \to \Omega$ is representable in Ω : it is well-known that for any set with at least 2 elements, such as Ω , no surjective mappings from the set onto its function space exist.

In practice, we do not distinguish functions from their representations; thus, we simply speak of *function* f , instead of *the function represented by* f . If we denote, as we do, ordinary function application by $\cdot$ too, the (notational) distinction even becomes void; formula (0) , for instance, then becomes:

(1) $(\mathbf{A}x :: F{\cdot}x = f{\cdot}x)$.

This formula shows that there is no need to introduce name F anymore.

convention 1.9.0: Expressions of the form $f \cdot x$ are called *applications*: we say that *function* f is *applied* to *argument* x .

□

Although Ω does not contain (representations of) all functions of type $\Omega \to \Omega$, the rules of the game have been chosen in such a way that Ω does represent a sufficiently interesting class of functions. For instance, considered as a set of functions, Ω is closed under function composition. Function composition can even be programmed in the basic formalism.

property 1.9.1: Ω contains a value c satisfying:

$$(\mathbf{A}\, f,g,x :: c \cdot f \cdot g \cdot x = f \cdot (g \cdot x))\ .$$

proof: This is easy: considered as an equation with unknown c , the above specification of c is admissible; hence, it suffices to encode its solution as an expression, giving:

$$c\ [\![\ c \cdot f \cdot g \cdot x = f \cdot (g \cdot x)\ ]\!]$$

□

corollary 1.9.2: Ω is closed under function composition; i.e:

$$(\mathbf{A}\, f,g :: (\mathbf{E}\, h :: (\mathbf{A}\, x :: h \cdot x = f \cdot (g \cdot x))))$$

□

For function f and value x , $f \cdot x$ is an element of Ω that may be interpreted as a function itself. Application of this function to another value y , say, yields $f \cdot x \cdot y$, and so on. Thus, such f may be considered as either a 2-argument function or a, so-called, *higher-order* function, or both simultaneously, if so desired. Thus, Ω not only represents a subset of $\Omega \to \Omega$, but also represents subsets of $\Omega \to (\Omega \to \Omega)$, and so on. Hence, in Ω , multi-argument or higher-order functions need no special treatment.

The specification of a function f often has the following form:

(2) $\quad (\mathbf{A}\, x : P \cdot x : Q \cdot x \cdot (f \cdot x))\ .$

Here P and Q are predicates on Ω and $\Omega \times \Omega$ respectively. Because (2) provides information about f·x only for those x that satisfy P·x , we call P the *domain* of f , or, if we wish to stress that it is a predicate, f's *precondition*; apparently, (2) expresses that we are only interested in f·x for x satisfying P·x . In particular, this is the case when we introduce some of the function's parameters as, so-called, *additional parameters* -- see chapter 4 -- ; usually, these parameters are redundant in the sense that the values they represent can also be expressed in the terms of the other parameters. In such cases, we use a precondition to fix the relation between the parameters of the function.

When, on a given domain, two functions have the same values, we call these functions *functionally equivalent* on that domain. This is captured by the following definition.

definition 1.9.3 (functional equivalence): For predicate P and functions f and g , "f and g are functionally equivalent on domain P" means:

$$(\mathbf{A}x : P{\cdot}x : f{\cdot}x = g{\cdot}x) \quad .$$

When it is clear from the context what domain is intended, we simply call f and g functionally equivalent.

□

Values that are functionally equivalent on a domain represent, on that domain, the same function. This does not mean, however, that functional equivalence implies equality: outside their domain, the two functions may have different values.

Usually, the domains of the functions we are interested in are proper subsets of Ω . Therefore, there is no point in introducing (3) as a postulate, for the simple reason that we cannot use it:

(3) $(\mathbf{A}f,g :: (\mathbf{A}x :: f{\cdot}x = g{\cdot}x) \Rightarrow f = g)$.

This property, often referred to as *extensionality*, is useless for our purposes, because, in order to apply it, we must prove $(\mathbf{A}x :: f{\cdot}x = g{\cdot}x)$; when f and g are supposed to have domain P , for P a proper subset of Ω , then $(\mathbf{A}x : P{\cdot}x : f{\cdot}x = g{\cdot}x)$ is about the strongest property we are both willing and able to prove about f and g .

The converse to (3) is:

(4) $(\mathbf{A} f,g :: (\mathbf{A} x :: f\cdot x = g\cdot x) \Leftarrow f = g)$.

This is an application of Leibniz's rule to function $\cdot$; it has nothing to do with extensionality. We often use it implicitly, as follows. Whenever we have derived a defining equation of the form $f\cdot x = F\cdot x$, for names f, x and expression F in which x does not occur, we may replace it by the definition $f = F$; phrased differently, definition $f = F$ is a correct implementation of $f\cdot x = F\cdot x$. This follows directly from (4) ; remember that definition $f\cdot x = F\cdot x$ is an abbreviation of $(\mathbf{A} x :: f\cdot x = F\cdot x)$.

1.10 Types

In this section, we extend the formalism with a, rather simple, notion of *types*.

definition 1.10.0 (type): A type is a subset of Ω . For type P and value x , we use the phrase "x has type P" as a synonym for $P\cdot x$.

□

As a result of our convention not to distinguish subsets of a set from predicates on that set, there is no formal difference between types and specifications. Hence, proving that an expression has a certain type is not different from proving that an expression satisfies a certain specification. Thus, no separate proof rules are needed to deal with types.

According to this definition, our notion of type is a semantic one, not a syntactic one. Consequently, "having a certain type" is not a mechanically decidable property. If we wish to assert that an expression has a certain type, this requires proof. Moreover, we cannot speak of *the* type of a value: generally, one and the same value may have different types, or, in other words, types need not be disjoint.

To ease the interpretation of elements of Ω as functions, as discussed in section 1.9, we introduce a *type constructor* that resembles the well-known constructor for function spaces from mathematics. Since types are predicates, the constructor actually is an operator on predicates.

definition 1.10.1 (the type constructor $\rightarrow$ ("to")): For types P and Q , the type $P \rightarrow Q$ is defined by:

$$(\mathbf{A}x :: (P \rightarrow Q) \cdot x \equiv (\mathbf{A}y : P \cdot y : Q \cdot (x \cdot y))) \ .$$

As an operator, $\rightarrow$ is right binding; its binding power equals that of $\Leftarrow$ and $\Rightarrow$.

□

convention 1.10.2: If x has type $P \rightarrow Q$, then x may be taken to represent a *function from* P *to* Q . In common parlance, we simply say that "x *is* a function of type $P \rightarrow Q$ ".

□

In ordinary mathematics $P \rightarrow Q$ denotes the set of *all* functions from P to Q . Here, it denotes the subset of Ω whose elements represent -- some of the -- functions from P to Q . Since we are only interested in functions that are representable in Ω , this restricted meaning of $\rightarrow$ does not harm us.

The following properties deal with introduction and elimination of $\rightarrow$.

property 1.10.3 ($\rightarrow$ introduction): For expression E , possibly containing x as free name, f a fresh name, and types P and Q :

$$(P \rightarrow Q) \cdot (f[\![f \cdot x = E]\!]) \equiv (\mathbf{A}x : P \cdot x : Q \cdot E)$$

□

property 1.10.4 ($\rightarrow$ elimination): For types P and Q :

$$(\mathbf{A}f, x :: P \cdot x \wedge (P \rightarrow Q) \cdot f \Rightarrow Q \cdot (f \cdot x) \ )$$

□

example 1.10.5: For types P, Q, R :

$I[\![I \cdot x = x]\!]$ has type $P \rightarrow P$

$K[\![K \cdot x \cdot y = x]\!]$ has type $P \rightarrow Q \rightarrow P$

$c[\![c \cdot f \cdot g \cdot x = f \cdot (g \cdot x)]\!]$ has type $(Q \rightarrow R) \rightarrow (P \rightarrow Q) \rightarrow (P \rightarrow R)$

every value has types $P \rightarrow \Omega$ and $\emptyset \rightarrow P$

□

example 1.10.6 (the range of a function): For function f, predicate R, defined by $R \cdot y \equiv (\mathbf{E}\, x :: y = f \cdot x)$, is the least type such that f has type $\Omega \rightarrow R$.

□

The following property resembles the rule for sequential programs according to which preconditions of programs may be strengthened and postconditions may be weakened. This is not so strange: to some extent, a function's argument and the function's value may be associated with the initial and final states of a sequential machine.

property 1.10.7 (monoticity properties of $\rightarrow$): For types P, Q, R and value x:

$$(P \subseteq Q) \wedge (Q \rightarrow R) \cdot x \Rightarrow (P \rightarrow R) \cdot x$$

$$(P \rightarrow Q) \cdot x \wedge (Q \subseteq R) \Rightarrow (P \rightarrow R) \cdot x$$

□

1.11 On recursion

We derive two theorems applicable to recursive definitions. These theorems are not deep, but they shed some light on how properties of recursively defined values can be proved. The theory developed here does not depend on properties of Ω; therefore, we present it in more general terms.

Let F be a function of type $V \rightarrow V$, where V is a set. Furthermore, all variables range over V, unless indicated otherwise.

We assume that f is a *fixed point* of F; i.e. f satisfies:

(0) $\quad f = F \cdot f$.

We investigate what we must prove about F in order that we may conclude that f satisfies a given specification. I.e. we are interested in theorems of the following form, in which R denotes f's specification and P is a condition to be satisfied by F:

(1) $\quad R \cdot f \Leftarrow P$.

Phrased differently, we investigate various solutions of equation (1) , in which P is the unknown.

Obviously, the weakest possible solution -- when (0) is all we know about f -- of (1) is given by:

(2) $(\mathbf{A}\, g : g = F{\cdot}g : R{\cdot}g)$.

In order to obtain more manageable forms, we strenghten (2) for the special case where R is given by:

(3) $(\mathbf{A}\, g :: R{\cdot}g \equiv (\mathbf{A}\, x : C{\cdot}x : Q{\cdot}x{\cdot}g)\ )$.

Here, Q is a predicate on $C \times V$ and we assume that C is a set that is equipped with a partial order $\leqslant$, such that $(C, \leqslant)$ is well-founded. So, we may use mathematical induction over C . We now have:

$$(\mathbf{A}\, x : C{\cdot}x : Q{\cdot}x{\cdot}f) \Leftarrow (2) \wedge (3) \quad .$$

We now derive, starting with (2) :

$$
\begin{array}{ll}
 & (\mathbf{A}\, g : g = F{\cdot}g : R{\cdot}g) \\
= & \quad \{ (3) \} \\
 & (\mathbf{A}\, g : g = F{\cdot}g : (\mathbf{A}\, x : C{\cdot}x : Q{\cdot}x{\cdot}g)\ ) \\
\Leftarrow & \quad \{ \text{ mathematical induction } \} \\
 & (\mathbf{A}\, g : g = F{\cdot}g : (\mathbf{A}\, x : C{\cdot}x : (\mathbf{A}\, y : C{\cdot}y \wedge y < x : Q{\cdot}y{\cdot}g) \Rightarrow Q{\cdot}x{\cdot}g)\ ) \\
= & \quad \{ \text{ Leibniz } \} \\
 & (\mathbf{A}\, g : g = F{\cdot}g : (\mathbf{A}\, x : C{\cdot}x : (\mathbf{A}\, y : C{\cdot}y \wedge y < x : Q{\cdot}y{\cdot}g) \Rightarrow Q{\cdot}x{\cdot}(F{\cdot}g)\ )\ ) \\
\Leftarrow & \quad \{ \text{ strengthening by weakening the range of the quantification } \} \\
 & (\mathbf{A}\, g :: (\mathbf{A}\, x : C{\cdot}x : (\mathbf{A}\, y : C{\cdot}y \wedge y < x : Q{\cdot}y{\cdot}g) \Rightarrow Q{\cdot}x{\cdot}(F{\cdot}g)\ )\ ) \\
= & \quad \{ \text{ unnesting dummies } \} \\
 & (\mathbf{A}\, g,x : C{\cdot}x : (\mathbf{A}\, y : C{\cdot}y \wedge y < x : Q{\cdot}y{\cdot}g) \Rightarrow Q{\cdot}x{\cdot}(F{\cdot}g)\ ) \quad .
\end{array}
$$

Thus, we have derived our first recursion theorem.

theorem 1.11.0 (first recursion theorem): (6) $\Leftarrow$ (4) $\wedge$ (5) , with:

(4) $f = F \cdot f$

(5) $(\mathbf{A}\, g,x : C \cdot x : (\mathbf{A}\, y : C \cdot y \wedge y < x : Q \cdot y \cdot g) \Rightarrow Q \cdot x \cdot (F \cdot g)\ )$

(6) $(\mathbf{A}\, x : C \cdot x : Q \cdot x \cdot f)$

□

By a (very) similar derivation, we obtain our second theorem for the even more special case where C is the natural numbers, with the usual ordering.

theorem 1.11.1 (second recursion theorem): (10) $\Leftarrow$ (7) $\wedge$ (8) $\wedge$ (9) , with:

(7) $f = F \cdot f$

(8) $(\mathbf{A}\, g : : Q \cdot 0 \cdot g)$

(9) $(\mathbf{A}\, g,i : 0 \leqslant i : Q \cdot i \cdot g \Rightarrow Q \cdot (i+1) \cdot (F \cdot g)\ )$

(10) $(\mathbf{A}\, i : 0 \leqslant i : Q \cdot i \cdot f)$

□

The step with hint "Leibniz" , in the above derivation, is rather arbitrary: we might have equally well replaced the other occurrence of g by $F \cdot g$, or we might have replaced g by $F^i \cdot g$, for some other natural i , different from 0 . The strengthening step following this step is directed towards simplification of the formula. Obviously, the freedom we have here indicates that many more theorems of this kind can be derived in a similar way.

In practice, we do not introduce name g ; since $f = F \cdot f$ is the only knowledge about f we have, the proof can be carried out in terms of f equally well. Moreover, very often we prove $(\mathbf{A}\, x : C \cdot x : Q \cdot x \cdot f)$ by mathematical induction straight away, without using the recursion theorems at all.

We conclude this section with the remark that the validity of the theorems derived above is independent of the question whether or not f , satisfying $f = F \cdot f$, does exist. Apparently, this is a separable concern.

1.12 Examples

We conclude this chapter with a few examples illustrating the use of the basic formalism. In this section, we use constants I , C , K of the previous sections. We recall their definitions here. Notice that, in these and the other definitions in this section, we use the syntactic form provided by the basic formalism -- without brackets ⟦ and ⟧ -- ; i.e. universal quantification over the parameters is left implicit:

$$I \cdot x = x \quad \& \quad C \cdot x = I \quad \& \quad K \cdot x \cdot y = x \quad .$$

example 1.12.0 (pair formation): Let pair, fst , and snd be defined by:

$$\begin{aligned} & \text{pair} \cdot x \cdot y \cdot s = s \cdot x \cdot y \\ \& \ & \text{fst} \cdot p \qquad = p \cdot K \\ \& \ & \text{snd} \cdot p \qquad = p \cdot C \quad . \end{aligned}$$

We then have:

```
    fst·(pair·x·y)
=        { unfolding fst }
    pair·x·y·K
=        { unfolding pair }
    K·x·y
=        { unfolding K }
    x  .
```

By a similar calculation we can derive that $\text{snd} \cdot (\text{pair} \cdot x \cdot y) = y$. Hence, a pair forming function and its inverses can be defined in the basic formalism. Thus, the product space $\Omega \times \Omega$ is representable in Ω .

□

example 1.12.1 (primitive boolean operations): Using functions pair, fst, snd defined in the previous example, we design constants true and false , and a function if with the following specification -- this specification implies that $\text{true} \neq \text{false}$ -- :

$$(\text{A} x,y :: \text{if} \cdot \text{true} \cdot x \cdot y = x \ \wedge \ \text{if} \cdot \text{false} \cdot x \cdot y = y) \quad .$$

Observing that $x = \text{fst} \cdot (\text{pair} \cdot x \cdot y)$, that $y = \text{snd} \cdot (\text{pair} \cdot x \cdot y)$, and that these two expressions are instances of the general expression $z \cdot (\text{pair} \cdot x \cdot y)$, we choose for if the following definition; this choice leaves us no further freedom for the definitions of true and false :

$$\begin{array}{lll} & \text{if}\cdot z\cdot x\cdot y & = z\cdot(\text{pair}\cdot x\cdot y) \\ \& & \text{true} & = \text{fst} \\ \& & \text{false} & = \text{snd} \quad . \end{array}$$

Now other elementary boolean functions can be defined, for example:

$$\begin{array}{lll} & \text{not}\cdot x & = \text{if}\cdot x\cdot \text{false}\cdot \text{true} \\ \& & \text{and}\cdot x\cdot y & = \text{if}\cdot x\cdot y\cdot \text{false} \\ \& & \text{or}\cdot x\cdot y & = \text{if}\cdot x\cdot \text{true}\cdot y \quad . \end{array}$$

□

These examples show that values defined to be used as functions may also be used as constants in the definitions of other functions. Here, we exploit that we need not distinguish between functions and other values.

example 1.12.2 (modularisation): Multiple occurrences of the same subexpression can be eliminated by means of where-clauses. This can even be done in two different ways. For example, for expression E and some binary operator $\oplus$, we derive:

$$\begin{array}{ll} & E \oplus E \\ = & \quad \{ \text{ let } f \text{ be a fresh name: property } 1.7.0.2 \ \} \\ & E \oplus E \ [\![f\cdot x = x \oplus x]\!] \\ = & \quad \{ \text{ folding } f \ \} \\ & f\cdot E \ [\![f\cdot x = x \oplus x]\!] \quad . \end{array}$$

And:

$$\begin{array}{ll} & E \oplus E \\ = & \quad \{ \text{ let } x \text{ be a fresh name: property } 1.7.0.2 \ \} \\ & E \oplus E \ [\![x = E]\!] \\ = & \quad \{ \text{ folding } x \text{ (twice) } \} \\ & x \oplus x \ [\![x = E]\!] \quad . \end{array}$$

The expression $x \oplus x \,[\![x = E]\!]$ is simpler than $f{\cdot}E \,[\![f{\cdot}x = x \oplus x]\!]$ and is, therefore, to be preferred. Elimination of such multiple occurrences may be necessary to simplify reasoning about such expressions, or to improve their efficiency, or both.

□

2 The program notation

2.0 Introduction

In this chapter we extend the functional-program notation with

- types Bool , Nat , and Int ,
- various operators,
- language constructs for case analysis and tuple formation.

The program notation, as presented in this monograph, has been chosen as sober as possible. We introduce only those notions that we consider necessary or convenient for the development and illustration of our programming techniques. Thus, we have not cared to think about the introduction of practical things such as, for example, *characters*, *strings*, or *enumeration types*. Even the notion of *tuples*, which provides a convenient framework for *recursive datatypes*, is introduced here in the simplest possible way.

2.1 On types and operators

In the following sections we introduce a few standard types and operators on these types. Here, we discuss in general terms how to use these operators properly. Furthermore, we introduce a syntactic convention by means of which these operators can be treated as expressions themselves.

Whenever we introduce a binary operator $\oplus$, the syntax of the notation must be extended accordingly with a rule:

$$\text{Exp} \rightarrow \text{'('}\ \text{Exp}\ \text{'}\oplus\text{'}\ \text{Exp}\ \text{')'} \quad .$$

In the following sections we do not reformulate this rule, over and over again, for each of the operators introduced. Instead, we only summarise the binding conventions used to reduce parentheses. Similarly, we omit the, equally obvious, syntax extensions corresponding to the unary operators.

Let $\oplus$ be a binary operator of type $P \times P \to Q$, for fixed types P and Q; i.e, for expressions E and F having type P, $E \oplus F$ is an expression having type Q. Notice, however, that $E \oplus F$ is a correct expression for *any* two expressions E and F: in order to conclude that $E \oplus F$ has type Q we must *prove* that E and F have type P. So, the syntactical form of the expression does not provide, all by itself, sufficient information on its type. Yet, if $\oplus$ "only" has type $P \times P \to Q$, then there is no point in using $\oplus$ with operands not having type P. One should keep in mind, however, that each use of such an operator brings along a proof obligation. In section 2.8 we give examples of the pitfalls caused by violation of this requirement.

For each binary operator $\oplus$ in the program notation we can construct an expression representing it, viz. $f \,[\![\, f \cdot x \cdot y = x \oplus y \,]\!]$. In this way, the operators from the notation can be treated as values themselves; they may, for instance, be used as arguments in function applications. For the sake of conciseness, we introduce an abbreviation for the above expression. Similarly, we introduce abbreviations for the functions obtained by fixing one of the two arguments.

definition 2.1.0: For binary operator $\oplus$ and expressions E and F, $(\oplus)$, $(E\oplus)$, and $(\oplus F)$ are expressions too; they satisfy:

$$\begin{aligned}
(\oplus) &= f \,[\![\, f \cdot x \cdot y = x \oplus y \,]\!] \\
(E\oplus) &= f \,[\![\, f \cdot y = E \oplus y \,]\!] \\
(\oplus F) &= f \,[\![\, f \cdot x = x \oplus F \,]\!]
\end{aligned}$$

□

warning 2.1.1: But for one exception, the new way of using parentheses introduced here causes no ambiguities. The exception is $(-F)$, in which $-$ now can be interpreted both as the unary and as the binary minus operator (introduced in section 2.3). By default, the interpretation as the unary operator is chosen.

□

example 2.1.2: $E \oplus F = (\oplus) \cdot E \cdot F$, $E \oplus F = (E\oplus) \cdot F$, and $E \oplus F = (\oplus F) \cdot E$.

□

convention 2.1.3: We use such abbreviations in the metanotation too. For example, $(=2)$ denotes predicate P with $P{\cdot}x \equiv (x=2)$, whereas $(=2) \cup (=3)$ denotes predicate P with $P{\cdot}x \equiv (x=2) \vee (x=3)$.

□

2.2 Function composition

In chapter 1 we have shown that function composition can be defined in the basic formalism. Here, we introduce a notation for it.

definition 2.2.0 (function composition): Function composition is denoted by the binary infix operator $\circ$ ("composed with") ; it satisfies:

$$(\mathbf{A}\, f,g,x :: (f\circ g){\cdot}x = f{\cdot}(g{\cdot}x)\ ) \quad .$$

$\circ$ binds weaker than $\cdot$, but both operators bind stronger that any of the other operators. $\circ$ is associative in the sense that $f\circ(g\circ h)$ and $(f\circ g)\circ h$ are functionally equivalent on domain Ω (cf. definition 1.9.3).

□

2.3 The types Bool, Nat, and Int

Informally, the sets Bool , Nat , and Int are given by:

definition 2.3.0:

$$\begin{aligned} \text{Bool} &= \{\,\text{true}\,,\text{false}\,\} \\ \text{Nat} &= \{\,0\,,1\,,2\,,3\,,\cdots\,\} \\ \text{Int} &= \text{Nat} \cup \{\,-1\,,-2\,,-3\,,\cdots\,\} \end{aligned}$$

□

Since these sets are well-known, we do not give formal definitions here. We incorporate them as types in the program notation by means of the following postulate.

postulate 2.3.1:

$$\begin{aligned} \mathrm{Bool} &\subsetneq \Omega \\ \mathrm{Nat} &\subseteq \Omega \\ \mathrm{Int} &\subseteq \Omega \end{aligned}$$

□

This postulate is realistic in the sense that it is possible to represent these types in the basic formalism by suitably designed expressions. The introduction of special nomenclature for (the elements of) these types is necessary, both for the sake of brevity and for the sake of *representational abstraction*.

Syntactically, the elements of these types are represented as suggested by definition 2.3.0 above. Furthermore, we extend the notation with the usual unary and binary operators on booleans and numbers. Their representations and their relative binding powers are given as follows.

summary 2.3.2 (syntax of the arithmetic and boolean operators):

In decreasing order of binding power, the operators are:

(0)	(unary) $-$
(1)	**max** , **min**
(2)	$*$, **div** , **mod**
(3)	$+$, $-$
(4)	(unary) $\neg$
(5)	$\wedge$, $\vee$
(6)	$\Leftarrow$, $\Rightarrow$
(7)	$\equiv$, $\not\equiv$

The operators in lines labelled (0) through (3) are called *arithmetic* operators; those in lines labelled (4) through (7) are called *boolean* operators. All operators except the unary ones are used in infix notation.

□

The meaning of these operators is the usual one. Since no universal consensus exists about the meaning of **div** and **mod** , we give our interpretation here. For the other operators we specify their types only.

summary 2.3.3 (semantics of the arithmetic and boolean operators):

(0) (unary) $-$ has type $\mathrm{Int} \to \mathrm{Int}$.

(1) **max** , **min** , $*$, $+$, $-$ have type $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Int}$; apart from $-$ they also have type $\mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}$.

(2) **div** has types $\mathrm{Int} \times \mathrm{Pos} \to \mathrm{Int}$ and $\mathrm{Nat} \times \mathrm{Pos} \to \mathrm{Nat}$, and **mod** has type $\mathrm{Int} \times \mathrm{Pos} \to \mathrm{Nat}$, where $\mathrm{Pos}{\cdot}x \equiv \mathrm{Nat}{\cdot}x \wedge 0 < x$. The pair $(a\,\mathbf{div}\,b\, ,\, a\,\mathbf{mod}\,b)$ is the (unique) solution of the equation $q,r:\ a = q * b + r \wedge 0 \leqslant r < b$, for a,b satisfying $\mathrm{Int}{\cdot}a \wedge \mathrm{Pos}{\cdot}b$.

(3) $\neg$ has type $\mathrm{Bool} \to \mathrm{Bool}$.

(4) $\wedge$, $\vee$, $\Leftarrow$, $\Rightarrow$, $\equiv$, $\not\equiv$ have type $\mathrm{Bool} \times \mathrm{Bool} \to \mathrm{Bool}$.

□

We assume the reader to be familiar with the algebraic properties of these operators. In calculations we mostly use these properties with no other justification than the general hint "algebra" or "calculus" .

convention 2.3.4: Expressions having type Bool , Nat , or Int are called *boolean*, *natural*, and *integer expressions* respectively.

□

2.4 The relational operators

An other important class of binary operators is formed by the *relational operators*.

summary 2.4.0 (relational operators):
The relational operators are $<, \leqslant, =, \neq, \geqslant, >$. They bind weaker than the arithmetic operators and stronger than the boolean operators. Their meaning is the usual one. They have types $\mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$ and, hence, also $\mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Bool}$.

□

Notice that the symbols $=$ and $\neq$ introduced above into the program notation also occur in the metanotation in which we discuss programs: in the program notation $E = F$ is an expression composed of subexpressions E and F and the operator $=$; in the metanotation, $E = F$ is the assertion that expressions E and F have the same value. The meanings of these two uses of $=$ are not completely the same. In the metanotation, $E = F$ is a well-defined boolean value for any two expressions E and F . In the program notation the value of $E = F$ is boolean only if both E and F have type Int ; in that case, the two meanings of $=$ coincide. For other expressions, the value of $E = F$ has been left unspecified, so we may not even conclude that its value is boolean.

In programs we use the equality operator only with integer expressions. So, in practice we need not be aware of the difference between the two uses of $=$; hence, the use of the same symbol for both purposes is harmless. One should keep in mind, however, that the use of $=$ and $\neq$ in programs brings along the same kind of proof obligation as the use of the other operators does.

2.5 Guarded selections

By means of, so-called, *guarded selections* we can use case analysis in programs. In contrast to other functional-program notations we have chosen a form in which the textual order of the, so-called, *alternatives* is irrelevant for the meaning of the construct. The advantage of this is that each alternative of a guarded selection can now be discussed in isolation: its meaning does not depend on its relative position within the construct. Moreover, the construct is symmetric. The notation chosen here strongly resembles the notation for *guarded commands* [Dij0].

definition 2.5.0 (syntax of guarded selections): A guarded selection is an expression formed according to the following rules:

```
Exp  →   '(' Gexp { '▯' Gexp } ')'
Gexp →   Exp '→' Exp  .
```

The elements of Gexp are called *alternatives*. In an alternative $B \rightarrow E$, expressions B and E are called the *guard* and the *guarded expression* respectively.

□

We stipulate that the semantics of guarded selections is captured sufficiently by the following postulate; i.e, it is all we need to be able to use guarded selections in programs.

postulate 2.5.1 (proof rule for guarded selections): For predicate R and guarded selection GS consisting of $n+1$ alternatives $B_i \rightarrow E_i$, $0 \leqslant i \leqslant n$, we have $R{\cdot}GS \Leftarrow (0) \wedge (1) \wedge (2)$, with:

(0) $(\mathbf{A}\, i :: Bool{\cdot}B_i)$

(1) $(\mathbf{E}\, i :: B_i)$

(2) $(\mathbf{A}\, i :: B_i \Rightarrow R{\cdot}E_i)$

□

Condition (0) in this rule states that all guards in a guarded selection must be boolean expressions. That this condition occurs in the semantical definition is a consequence of our semantical interpretation of types. Condition (1) states that at least one of the guards must have value true , and condition (2) states that all guarded expressions whose guards are true must satisfy the specification of the whole construct. Notice that in (1) and (2) we have written B_i instead of $B_i = true$; in view of (0) this is correct.

The above proof rule does not specify the value of a guarded selection completely. As explained in section 1.0, this does not mean that we propose a nondeterministic notation. As any other expression, a guarded selection has a single, uniquely determined, value that is, however, only partially specified by the proof rule.

Guarded selections can be easily implemented in the basic formalism, even in several ways. For example, expression $(B_0 \rightarrow E_0 \, [\!] \, B_1 \rightarrow E_1 \, [\!] \, B_2 \rightarrow E_2)$ may be encoded as $if{\cdot}B_0{\cdot}E_0{\cdot}(if{\cdot}B_1{\cdot}E_1{\cdot}E_2)$, where if is the function defined in example 1.12.1: this encoding satisfies postulate 2.5.1. Each of these implementations is, however, overspecific in the sense that it contains more information on the value of the expression than we care to know. In order to avoid this, we have defined guarded selections in the above way.

example 2.5.2: Let $E23$ be the expression $(true \rightarrow 2 \, [\!] \, true \rightarrow 3)$. Conditions (0) and (1) of the proof rule are satisfied. Moreover, condition (2) amounts to $R{\cdot}2 \wedge R{\cdot}3$, for any predicate R . The strongest R satisfying this is, of course, given by $R{\cdot}x \equiv (x=2) \vee (x=3)$. Hence, we conclude

$E23 = 2 \vee E23 = 3$. If we take (=2) for R , condition (2) is false; so, the proof rule gives no information on whether or not $E23 = 2$. Generally, we are not able to derive, by means of the proof rule, $E23 = x$, for any x . In particular, we do not know whether or not (true → 2 ▯ true → 3) equals (true → 3 ▯ true → 2) .

□

The above example shows that the value of a guarded selection may depend on the textual order of its alternatives. The proof rule itself, however, is symmetric with respect to this textual order. I.e, if we have proved, by means of rule 2.5.1, that a guarded selection satisfies a given specification, then every guarded selection obtained by permutation of its alternatives also satisfies that specification. To all intents and purposes, this is sufficient.

example 2.5.3: With respect to any specification, expressions (true → 2 ▯ true → 3) and (true → 3 ▯ true → 2) are equivalent.

□

For specifications of the form (=X) , for some fixed value X , rule 2.5.1 can be instantiated as follows.

property 2.5.4 (special case of postulate 2.5.1): For value X and guarded selection GS consisting of $n+1$ alternatives $B_i \rightarrow E_i$, $0 \leq i \leq n$, we have $GS = X \Leftarrow (0) \wedge (1) \wedge (2)$, with:

(0) $(\mathbf{A}\, i :: \text{Bool}{\cdot}B_i)$

(1) $(\mathbf{E}\, i :: B_i)$

(2) $(\mathbf{A}\, i :: B_i \Rightarrow E_i = X)$

□

This rule shows that, in order to prove that a guarded selection has value X , we must prove that each alternative with a true guard equals X .

example 2.5.5:

$(a\leq b \to a \;[\!]\; b\leq a \to b) = X$

$\Leftarrow$ { rule 2.5.4 }

$\mathrm{Bool}\cdot(a\leq b) \wedge \mathrm{Bool}\cdot(b\leq a) \wedge (a\leq b \vee b\leq a) \wedge (a\leq b \Rightarrow a=X) \wedge (b\leq a \Rightarrow b=X)$

$\Leftarrow$ { $\leq$ has type $\mathrm{Int}\times\mathrm{Int} \to \mathrm{Bool}$ (twice) }

$\mathrm{Int}\cdot a \wedge \mathrm{Int}\cdot b \wedge (a\leq b \vee b\leq a) \wedge (a\leq b \Rightarrow a=X) \wedge (b\leq a \Rightarrow b=X)$

$=$ { for $\mathrm{Int}\cdot a \wedge \mathrm{Int}\cdot b$: $a\leq b \vee b\leq a$, definition of **min** }

$\mathrm{Int}\cdot a \wedge \mathrm{Int}\cdot b \wedge a\,\mathbf{min}\,b = X$.

Thus, we have derived: $\mathrm{Int}\cdot a \wedge \mathrm{Int}\cdot b \Rightarrow (a\leq b \to a \;[\!]\; b\leq a \to b) = a\,\mathbf{min}\,b$.

In practice $\mathrm{Int}\cdot a \wedge \mathrm{Int}\cdot b$ will be part of the context in which the derivation is carried out, and it will be used tacitly. Notice, however, that the properties of $\leq$ and **min** used in the last step of the above derivation are only valid for integer a and b .

□

The following property of guarded selections can be expressed verbally by saying that, under certain conditions, function application distributes from the left over the alternatives of a guarded selection. We use it quite often, without explicit reference, to clean up the code of our programs.

property 2.5.6: For function F , value X and guarded selection GS consisting of $n+1$ alternatives $B_i \to E_i$, $0\leq i\leq n$, we have $(3) \Leftarrow (0) \wedge (1) \wedge (2)$, with:

(0) $(\mathbf{A}\,i :: \mathrm{Bool}\cdot B_i)$

(1) $(\mathbf{E}\,i :: B_i)$

(2) $(\mathbf{A}\,i :: B_i \Rightarrow F\cdot E_i = X)$

(3) $F\cdot GS = (B_0 \to F\cdot E_0 \;[\!]\; \cdots \;[\!]\; B_n \to F\cdot E_n)$.

proof: Assuming $(0)\wedge(1)\wedge(2)$ we prove (3) by showing that both sides of the equation are equal to X . First, $F\cdot GS = X$ on account of postulate 2.5.1, with $R\cdot x \equiv F\cdot x = X$; the premisses of 2.5.1 then amount to $(0)\wedge(1)\wedge(2)$. Second, $(B_0 \to F\cdot E_0 \;[\!]\; \cdots \;[\!]\; B_n \to F\cdot E_n) = X$ on account of property 2.5.4, the premisses of which also amount to $(0)\wedge(1)\wedge(2)$.

□

2.6 Tuples

By means of *tuple formation* or, for short, *tupling*, any (finite) number of values can be treated as a single, composite value. Algebraically, this means that by means of tupling Ω^n is representable within Ω, for natural n. Syntactically, tuples are defined as follows.

definition 2.6.0 (syntax of tuples): A *tuple* is an expression formed according to the following rules:

```
Exp  → '[' Exps ']'
Exps → Empty
Exps → Exp { ',' Exp } .
```

Thus, a tuple is a sequence of expressions separated by commas and embraced by [and] . For sequences of length n, the tuples thus formed are called n-tuples . The expressions occurring in a tuple are called the tuple's *elements*. The, one and only, 0-tuple [] is also called (the) *empty* (*tuple*).

□

In order that, for m and n such that $m \neq n$, m-tuples and n-tuples can be distinguished, we introduce the following operator.

definition 2.6.1 (the size operator): The prefix operator # ("size of") is defined as follows:

for natural n and n-tuple x : $\#x = n$.

□

corollary 2.6.2: For m-tuple x and n-tuple y , we have: $m \neq n \Rightarrow x \neq y$.

□

Furthermore, an n-tuple is a function on the natural numbers less than n ; the values of this function are the tuple's elements. Thus, for any tuple we can form expressions denoting the tuple's elements. This is captured by the following postulate.

postulate 2.6.3 (element selection): For natural n, expressions E_i, $0 \leqslant i < n$, we have:

$$(\mathsf{A}\, i : 0 \leqslant i < n : [E_0, \cdots, E_{n-1}]\cdot i = E_i\,) \quad .$$

□

Actually, in our program notation, there is no difference between tuples and *finite lists*, as introduced, and defined more formally in chapter 5.

2.7 Parameter and definition patterns

Guarded selections are used to encode case analysis in the program notation. Their use gives rise to rather long formulae, which is awkward if we wish to manipulate one of the expression's alternatives only; in that case we prefer to manipulate that alternative in isolation. To ease this, we provide the notation with some syntactical sugar by means of which the use of guarded selections in definitions can largely be avoided. Instead of writing one definition with a long guarded selection for its defining expression, we use a multiple definition with as many components as there are alternatives. Furthermore, we introduce abbreviations for often occurring forms of definitions.

definition 2.7.0 (multiple definition of a single name): For expressions B_i and E_i, $0 \leqslant i \leqslant n$, name x, and parameter list pp, definition (1) may be written as (0), with:

$$\begin{array}{lrl} (0) & & x\ pp = (B_0 \rightarrow E_0) \\ & \& & x\ pp = (B_1 \rightarrow E_1) \\ & & \vdots \\ & \& & x\ pp = (B_n \rightarrow E_n) \end{array}$$

$$(1) \qquad x\ pp = (\,B_0 \rightarrow E_0 \,[\!]\, B_1 \rightarrow E_1 \,[\!]\, \cdots \,[\!]\, B_n \rightarrow E_n\,)$$

□

At first sight, the use of a multiple definition for a single name seems awkward: for $n+1$ alternatives, the left-hand side of the definition must now be repeated n times. We can now, however, introduce the possibility to write such definitions in a form that resembles the way in which we write down *recurrence relations* very much. Thus, encoding a set of recurrence relations as definitions in the program notation now turns out to be a trivial operation in the majority of cases. We call the syntactic forms used for this purpose *parameter patterns*. They can be used in definitions when the parameters of the function defined are supposed to have certain, fixed, types. Here, we define such patterns for naturals and for tuples. In chapter 5 we also introduce patterns for lists.

definition 2.7.1 (definition and parameter patterns for Nat): For expression E, natural c, name n, and fresh name x, we introduce the following abbreviations for definitions:

```
      n+c = E      means     n = E - c
      f·c = E      means    f·x = ( x = c  →  E )
  f·(n+c) = E      means    f·x = ( x ≥ c  →  (E |[ n+c = x ]|) )
```

□

example 2.7.2: Two equivalent encodings of the factorial function are:

```
    fac |[ fac·n = ( n = 0  →  1
                   [] n ≥ 1  →  n * fac·(n-1)
                   )
       ]|
```

and:

```
    fac |[ fac·0      = 1
       & fac·(n+1)  = (n+1) * fac·n
       ]|  .
```

Similarly, an expression for the function whose values are the Fibonacci numbers is:

```
    fib |[ fib·0       = 0
       & fib·1       = 1
       & fib·(n+2)  = fib·(n+1) + fib·n
       ]|
```

□

Tuples are of interest only because of their elements. Therefore, we introduce a shorthand notation for the introduction of names for the elements of a tuple, without explicit use of element selection and without naming the tuple itself. This is useful both for naming the elements of a parameter that is supposed to be a tuple and for interpreting an arbitrary expression as a tuple.

definition 2.7.3 (definition and parameter patterns for tuples): Here, we give a definition for the empty tuple and for 3-tuples only; this is sufficiently representative for the general case. For expression E , names a, b, c , and fresh name x , we have:

$$\begin{array}{rll} [a,b,c] = E & \text{means} & a = E{\cdot}0 \ \& \ b = E{\cdot}1 \ \& \ c = E{\cdot}2 \\ f{\cdot}[\,] = E & \text{means} & f{\cdot}x = (\,\#x = 0 \rightarrow E\,) \\ f{\cdot}[a,b,c] = E & \text{means} & f{\cdot}x = (\,\#x = 3 \rightarrow (E[\![\, [a,b,c] = x \,]\!])\,) \end{array}$$

□

Because of the use of guarded expressions in this definition, we can construct multiple definitions, with parameter patterns, for functions taking tuples of various sizes for their arguments. We give an example of this in the next section.

2.8 Examples

In this section we give three examples. The first example shows how the arithmetic operators should *not* be used. The next two examples illustrate the use of tuples. Here, we present a few simple programs, without derivations. The use of *tupling*, as a programming technique, is discussed more extensively in chapter 4.

example 2.8.0: We consider the expression $x[\![x = 2-x]\!]$. In order to compute its value, we might derive:

$$\begin{array}{ll} & x = 2-x \\ = & \quad \{ \text{ algebra } \} \\ & x = 1 \quad , \end{array}$$

Thus, we might conclude that $x\,|[\, x = 2-x \,]| = 1$. Drawing this conclusion is, however, wrong, because the above derivation is wrong: the hint "algebra" refers to algebraic properties of the minus operator that are only valid for integer x . A correct derivation is:

$$\begin{array}{ll} & x = 2-x \wedge \mathrm{Int}\cdot x \\ = & \quad \{\ \text{algebra}\ \} \\ & x = 1 \wedge \mathrm{Int}\cdot x \\ = & \quad \{\ \mathrm{Int}\cdot 1\ \} \\ & x = 1 \ . \end{array}$$

In order to conclude that $x\,|[\, x = 2-x \,]| = 1$, we should also prove -- on account of the proof rule for where-clauses -- : $(\mathbf{A}\, x :: x = 2-x \Rightarrow \mathrm{Int}\cdot x)$. This, however, is impossible.

Similarly, the equation $x\colon x = 1+x$ is an admissible equation; hence, $x\,|[\, x = 1+x \,]|$ is a correct expression. This equation has, however, no integer solutions. Hence, $\Omega \neq \mathrm{Int}$. Thus, we conclude that $x\,|[\, x = 1+x \,]|$ has a non-integer value; moreover, we can not refute (nor prove) that this value is also a solution of the equation $x\colon x = 2-x$.

□

example 2.8.1: We give three, equivalent, expressions for function fip , satisfying: $(\mathbf{A}\, n : \mathrm{Nat}\cdot n : \mathrm{fip}\cdot n = [\, \mathrm{fib}\cdot n\, ,\, \mathrm{fib}\cdot(n+1)\,]\,)$, where fib is the function defined in example 2.7.2. Conversely, we have: $(\mathbf{A}\, n : \mathrm{Nat}\cdot n : \mathrm{fib}\cdot n = \mathrm{fip}\cdot n\cdot 0\,)$.

$$\begin{array}{lll} (0) & \mathrm{fip}\ |[\ \mathrm{fip}\cdot 0 & = [0,1] \\ & \quad \&\ \mathrm{fip}\cdot(n+1) & = g\cdot(\mathrm{fip}\cdot n)\ |[\ g\cdot[a,b] = [b,a+b]\ ]| \\ & \quad]| & \end{array}$$

$$\begin{array}{lll} (1) & \mathrm{fip}\ |[\ \mathrm{fip}\cdot 0 & = [0,1] \\ & \quad \&\ \mathrm{fip}\cdot(n+1) & = [\ x\cdot 1\ ,\ x\cdot 0 + x\cdot 1\ ]\ |[\ x = \mathrm{fip}\cdot n\ ]| \\ & \quad]| & \end{array}$$

$$\begin{array}{lll} (2) & \mathrm{fip}\ |[\ \mathrm{fip}\cdot 0 & = [0,1] \\ & \quad \&\ \mathrm{fip}\cdot(n+1) & = [b,a+b]\ |[\ [a,b] = \mathrm{fip}\cdot n\ ]| \\ & \quad]| \ . & \end{array}$$

In (0) we used an auxiliary function g ; in its definition we used a parameter pattern to introduce names for the elements of its parameter. In (2) we used a definition pattern to introduce names for the elements of (tuple) fip·n .

□

example 2.8.2 (recursive datatypes): A possible way to define, so-called, *labelled binary trees* is: a tree either is *empty* or consists of a *value* and two *(sub)trees*. Such trees can be represented by expressions, as follows. The empty tree is represented by the empty tuple, [] , and a non-empty tree is represented by a triple [a,s,t] , where a is the value and s and t are the subtrees of the tree. Now, all sorts of functions on trees can be defined; we just give a few examples, without further comments.

the *number of nodes* of a tree:

```
siz |[ siz·[]       = 0
     & siz·[a,s,t]  = 1 + siz·s + siz·t
     ]|   .
```

the *height* of a tree:

```
hgt |[ hgt·[]       = 0
     & hgt·[a,s,t]  = 1 + hgt·s max hgt·t
     ]|   .
```

extension of a tree with an new node (one of many possibilities):

```
ext |[ ext·a·[]      = [a , [] , [] ]
     & ext·a·[b,s,t] = [a , ext·b·t , s]
     ]|   .
```

removal of the root of a non-empty tree:

```
rem |[ rem·[a,[],s]        = s
      & rem·[a,[b,u,v],s] = [b,s,t] |[ t = rem·[b,u,v] ]|
      ]|   .
```

Notice that, in the last example, we have used parameter patterns in a *nested* way. By means of mathematical induction -- e.g. on the size of trees -- various properties of these functions can be proved; for example, for tree s and value a :

    siz·(ext·a·s) = 1 + siz·s , and
    rem·(ext·a·s) = s .

□

3 On efficiency

3.0 Introduction

The execution of a computer program requires computation time and storage space. The efficient use of these resources is a major concern in programming [Hoa0]. Although space and time can, to some extent, be traded against each other, this freedom is usually exploited in one direction only: in order to save computation time additional storage space is used.

In order to be able to discuss the efficiency of a program, we need rules by means of which the amount of time and space used during program execution can be determined from the program text. Because these rules depend on how programs are executed, we must make some assumptions about program execution first.

In this monograph we hardly pay attention to the space requirements of our programs. It is rather difficult to formulate general rules to determine a functional program's time and space requirements without an elaborate study of how functional programs can be executed. Such a study, however, exceeds the scope of this monograph. Therefore, we confine our attention to the execution times of our programs, and we do so in an informal way only. Fortunately, for the purpose of designing efficient programs, this suffices.

This chapter consist of two parts. First, we develop a rather simple *computational model* for our program notation. Second, using this model, we formulate a few rules by means of which the *time complexity* of a large class of programs can be determined. For a more extensive account of the problems associated with the execution of functional programs, see [Pey].

3.1 Evaluation of expressions

According to definition 1.5.2, a (functional) program is an expression without free names; in this chapter we use "expression" for "expression without free names". Execution of this expression is computation of a *suitable representation* of its value, a process that is also called *evaluation* of that expression. The question now arises what a suitable representation of the expression's value is and how it can be computed.

One possible representation of an expression's value is, of course, the expression itself. Although this would make evaluation a trivial operation, this representation is not suitable: we require that each value be represented *uniquely*; i.e, all expressions with the same value should yield, when evaluated, the same representation of that value. Apart from this requirement, there are no objections against using expressions to represent values. In order to obtain a unique representation, we select from each class of expressions having the same value a, so-called, *canonical expression* to represent the common value of all expressions belonging to that class. We call the canonical expression representing the value of an expression *the canonical form* of that expression. Preferably, canonical expressions are simple; for example, as a representation of the number 5 , expression 5 is to be preferred to the, somewhat arbitrary, expression $2+3$.

So, execution of a program amounts to computation of its canonical form. Obviously, this is only possible if the canonical expressions have been chosen in such a way that the canonical form of each program is effectively computable. Our program notation, however, will also contain -- as any other nontrivial program notation: vide the *halting problem* -- programs that may give rise to nonterminating computations. Hence, it is impossible to assign computable canonical forms to *all* programs, but this is not necessary either: we obtain a useful program notation if we are able to assign canonical forms to the programs in a nontrivial subset of the set of all programs. We call expressions that do have canonical forms *normal expressions*.

3.2 Canonical expressions

We define canonical expressions in such a way that all boolean and integer expressions have canonical forms. Moreover, we want all tuples -- and, hence, all so-called *finite lists*, as introduced in chapter 5 -- to have canonical forms, whenever their elements have canonical forms. Within the scope of this monograph such a modest set of canonical expressions is sufficient.

definition 3.2.0 (canonical expressions): The canonical expressions are the elements of the syntactic category Cexp , defined as follows:

```
Cexp  → Bool
Cexp  → Int
Cexp  → '[' Cexps ']'
Cexps → Empty
Cexps → Cexp { ',' Cexp }
```

In this definition, Bool and Int denote the syntactic categories of the boolean and integer constants respectively. In words, this definition states that the canonical expressions are the boolean and integer constants and the tuples formed from canonical expressions.

□

Canonical expressions do not contain names nor where-clauses. As a result, they do not contain free names either; hence, canonical expressions are programs.

example 3.2.1: Here are a few canonical expressions:

```
5
true
[]
[5,true]
[[],5,true,[5,true]]
[2,3,5,7,11,13,17,19,23,29]
[5,[4,[2,[],[]],[0,[],[]]],[3,[1,[],[]],[]]]    .
```

The last of these expressions represents the value of expression ext·5·(ext·4·(ext·3·(ext·2·(ext·1·(ext·0·[]))))) , where ext is the function defined in example 2.8.2.

□

The requirement that the canonical representation of a program be unique implies that all canonical expressions must have different values. Because this does not follow from the postulates in the previous chapters, we impose this requirement explicitly.

postulate 3.2.2: (Syntactically) different canonical expressions have different values.

□

3.3 Reduction

In general terms, an expression can be evaluated in the following way. If the expression is a canonical expression -- this is syntactically decidable -- evaluation terminates. If it is not a canonical expression it is transformed, by application of some manipulation rule, into another expression with the same value. Then, the same evaluation process is applied to the new expression.

This evaluation process need not terminate, but when it does the right canonical form is produced; this follows from the invariant that the expression is replaced by expressions with the same value, and from the property that all expressions with the same value have the same canonical form. Of course, evaluation will certainly not terminate when the expression has no canonical form. If it has a canonical form, termination depends on *how* the expression is manipulated. We return to this later. First, we discuss a number of ways to transform expressions into equivalent expressions.

Canonical expressions consist of boolean and integer constants and of tuple formation brackets only. They do not contain operators, names, where-clauses, and guarded selections; hence, during evaluation of an expression these must be eliminated.

Boolean operators can be meaningfully applied to boolean operands only. Boolean expressions have canonical forms; hence, a boolean operator can be eliminated by, first, evaluating its operands to canonical form, and, second, replacing the expression thus obtained by (the boolean constant representing) its value. Similarly, arithmetic and relational operators can be dealt with. The only operator not fitting into this pattern is · ; the only way to eliminate it is by unfolding (in combination with names) or by application of postulate 2.6.3 (element selection in tuples).

The overall structure of a program is an expression followed by zero or more where-clauses. The value of the program is the value of the expression, where the where-clauses provide definitions for all -- by definition 1.5.2 -- names occurring in the expression. The only way to eliminate these names is by unfolding (cf. rule 1.7.1.0). For a definition of the form $x \cdot y \cdot z = E$, unfolding x is only possible if x occurs in an application $x \cdot A \cdot B$. Replacement of $x \cdot A \cdot B$ by $E(y,z \leftarrow A,B)$ eliminates one occurrence of x and two $\cdot$'s , but the expression $E(y,z \leftarrow A,B)$ may contain, of course, further occurrences of x , $\cdot$, or other names. Hence, the use of unfolding does not guarantee progress.

A where-clause may be omitted from the expression if no name defined in the where-clause occurs in the expression -- where-clause elimination, cf. property 1.7.0.2 -- .

The proof rule for guarded selections (rule 2.5.1) gives information about the value of a guarded selection only when its guards are boolean expressions. In that case, a guarded selection can be evaluated, first, by evaluating its guards until a guard is encountered with value true , and, second, by replacing the guarded selection by the guarded expression corresponding to this guard. Because all guards are boolean, and because of the symmetry of the proof rule, the order in which the guards are evaluated is irrelevant, but, in order to keep the computation deterministic, this order should always be the same.

Both folding and unfolding are transformations that do not affect the value of the expression. It is typical for functional program notations that, when it comes to evaluation, only unfolding is used. Because of this restriction, the process of evaluation is also called *reduction*: unfolding is the direct counterpart in our notation of *β-reduction* in λ-calculus.

Generally, expressions can be reduced in many ways. This freedom allows a choice among many different, so-called, *reduction strategies*. As stated above, independent of the reduction strategy used, evaluation of an expression either does not terminate or yields the canonical form of that expression. Moreover, a well-known result from λ-calculus is the second Church-Rosser Theorem -- or generalisations thereof [Hin] -- , which states that a reduction strategy exists, called *normal order reduction*, that terminates for every normal expression.

3.4 Lazy evaluation

The reason why not all reduction strategies terminate for all normal expressions is that a normal expression may have subexpressions that are not normal themselves. In that case, evaluation of such a subexpression does not terminate. Hence, the evaluation strategy must manipulate the subexpressions of the program in a very controlled way.

example 3.4.0: The only transformation applicable to $x \;[\![\; x = x \;]\!]$ is unfolding x ; this yields the same expression again. Moreover, this expression is not canonical; so, we conclude that it has no canonical form. Evaluation of this expression will not terminate, whatever reduction strategy is used. Now we consider the expression $f{\cdot}x \;[\![\; x = x \;\&\; f{\cdot}y = 5 \;]\!]$. By unfolding f , the expression can be reduced to 5 , which is canonical, whereas unfolding x gives rise to a nonterminating computation.

□

example 3.4.1: We consider expression $x{\cdot}0 \;[\![\; x = [5,x] \;]\!]$. On the one hand, repeated unfolding of x again leads to a nonterminating computation; on the other hand, the following computation is possible too:

$x{\cdot}0$

= { unfolding x }

$[5,x]{\cdot}0$

= { element selection }

5 .

□

The latter example shows that it is sometimes necessary to evaluate a subexpression only *partially*: the subexpression must not be evaluated completely, because this may be impossible; yet, it must be subjected to some evaluation steps in order to make the expression it is part of amenable for reduction.

Normal order reduction has the property that subexpressions are only evaluated as far as necessary for the evaluation of the expression they are part of. Therefore, normal order reduction is also called or *demand-driven evaluation* or *lazy evaluation* [Pey].

3.5 Sharing

We consider expression $x + x \,[\![\, x = E \,]\!]$, for some expression E not containing x . Unfolding both occurrences of x and subsequent where-clause elimination yields $E + E$. The latter expression contains the same subexpression, E , twice. Evaluation of $E + E$ amounts to evaluating E twice followed by addition of the two values thus obtained: we do not expect the evaluating mechanism to *recognise* that the two subexpressions happen to be the same. On the other hand, occurrences of the *same* name in an expression, such as the x's in $x + x$, explicitly refer to the *same* value; therefore, it would be reasonable to expect that in such cases the expression bound to that name is evaluated *at most once*. This can be achieved by manipulating the expressions occurring in the where-clause. If, for instance, the canonical form of E is 5 , the computation might proceed as follows:

$$
\begin{array}{ll}
 & x + x \,[\![\, x = E \,]\!] \\
= & \quad \{ \text{ evaluate E first } \} \\
 & x + x \,[\![\, x = 5 \,]\!] \\
= & \quad \{ \text{ unfolding x (twice) } \} \\
 & 5 + 5 \,[\![\, x = 5 \,]\!] \\
= & \quad \{ \text{ et cetera... } \} \\
 & 10 \quad .
\end{array}
$$

This is called *sharing*: multiple occurrences of the same name give rise to a single evaluation of the expression corresponding to that name, namely as soon as evaluation of that expression is required. Without going into the details of how this is implemented, we assume that the use of names bound to expressions by means of where-clauses allows shared evaluation of these expressions. Well-known elaborations of this idea are, so-called, *environment-based* and *graph-reduction* evaluators [Pey].

The same phenomenon occurs with the unfolding of functions with parameters. This involves substitution, in the function's defining expression, of the arguments for its parameters. Thus, for each occurrence of the same parameter a copy of the argument expression is substituted, which again may cause multiple evaluation of the same expression. Moreover, substitution is a laborious operation. Substitution, and its disadvantages, can be avoided as follows. For expressions E and F and name y not occurring in F, we have -- cf. example 1.7.1.1 -- : $E(y \leftarrow F) = E|[y = F]|$. Hence, the expression $x \cdot F|[x \cdot y = E]|$ equals $E|[y = F]||[x \cdot y = E]|$. In this way, unfolding can be implemented without the use of substitution. The where-clauses, such as $|[y = F]|$, introduced by this transformation lend themselves for sharing as discussed above. Therefore, we also assume that the argument expressions in function applications are evaluated at most once.

3.6 The time complexity of expressions

The exact amount of time needed to execute a program strongly depends on details of the particular implementation used. As is usual in discussions about programming, we consider these details as irrelevant; therefore, we confine ourselves to the *time complexity* of our programs.

In the case of functional programming, we are often interested in the design of (a definition of) a function; i.e, the program is intended to be used in function applications (in other programs), it is not intended to be executed itself. The time complexity of this program then is a function with the same domain as the function represented by the program: for function F , the time needed to evaluate the application F·E usually depends on the value of E . So, with F we associate a function TF with the interpretation that TF·x denotes the amount of time needed to evaluate F·x . Here, we adopt the convention that TF·x does *not* comprise the evaluation of the argument supplied for x . Hence, the amount of time needed to evaluate F·E is TF·E + "the time needed to evaluate E (as far as needed)".

For the sake of brevity, we call "the amount of time needed to evaluate E" simply *the cost of E*. Notice that this is a function on the set of expressions, not on the set of their values: different expressions with the same value may have different costs. Thus, we have: "the cost of F·E" = TF·E + "the cost of E".

We now discuss how to determine TF for given program F . We assume that programs are executed with lazy evaluation and sharing. We are only interested in the time complexity of our programs, and we assume the boolean, arithmetic, and relational operators to have O(1) time-complexity. Therefore, we use the following, abstract notion of cost: in the process of evaluation, we only count the number of unfoldings. We stipulate that unfolding is the essential operation in the evaluation process and that the number of unfoldings used is sufficiently representative for an expression's cost.

definition 3.6.0 (cost of an expression): The *cost of an expression* is the minimal number of unfoldings needed to evaluate it.

□

definition 3.6.1 (time complexity of a function): For function F , the *time complexity of F* is the function TF with, for value x :

TF·x = "the cost of F·x" .

I.e. TF·x is the minimal number of unfoldings needed for evaluation of F·x , not counting the cost of x .

□

Recursive definitions of functions give rise to, so-called, recurrence relations for their time complexities. By solving these recurrences relations, by means of "ordinary" mathematical techniques, we obtain explicit characterisations of a function's time complexity.

3.7 Examples

We discuss the time complexities of a few examples.

example 3.7.0: We consider function fac defined by

```
   fac·0      = 1
 & fac·(n+1) = (n+1) * fac·n    .
```

With Tfac for fac's time complexity, we obtain recurrence relations for Tfac in the following way. Evaluation of fac·0 requires unfolding fac once; hence Tfac·0 = 1 . Evaluation of fac·(n+1) requires unfolding fac once, giving (n+1) * fac·n ; evaluation of this expression involves evaluation of fac·n , which requires Tfac·n unfoldings. Hence, Tfac·(n+1) = 1 + Tfac·n , for n , $0 \leqslant n$. Thus, we obtain:

$$\begin{aligned} Tfac\cdot 0 &= 1 \\ Tfac\cdot(n+1) &= 1 + Tfac\cdot n \ , \ 0 \leqslant n \quad . \end{aligned}$$

The solution to these recurrence relations is $Tfac\cdot n = 1 + n$, for natural n .
□

In this example, we have not taken into account the use of parameter patterns in the function's definition. In chapter 2 we have defined parameter patterns by means of additional names defined in where-clauses. In the above example, we have, for the sake of simplicity, not counted the number of unfoldings needed to eliminate these additional names. This is harmless, and throughout this monograph we shall ignore parameter patterns in efficiency considerations.

example 3.7.1: We consider function fib defined by

$$\begin{aligned} & fib\cdot 0 &&= 0 \\ \& \ & fib\cdot 1 &&= 1 \\ \& \ & fib\cdot(n+2) &&= fib\cdot(n+1) + fib\cdot n \quad . \end{aligned}$$

With Tfib for fib's time complexity, we obtain from this definition the following recurrence relations:

$$\begin{aligned} Tfib\cdot 0 &= 1 \\ Tfib\cdot 1 &= 1 \\ Tfib\cdot(n+2) &= 1 + Tfib\cdot(n+1) + Tfib\cdot n \ , \ 0 \leqslant n \quad . \end{aligned}$$

The solution to these relations is $Tfib\cdot n = 2 * fib\cdot(n+1) - 1$, for $n, 0 \leqslant n$; so, we have: $Tfib\cdot n$ is $O(fib\cdot n)$.
□

example 3.7.2: We consider function fip defined by

$$\begin{aligned} & fip\cdot 0 &&= [0,1] \\ \& \ & fip\cdot(n+1) &&= g\cdot(fip\cdot n) \ [\![\ g\cdot[a,b] = [b,a+b] \ ]\!] \quad . \end{aligned}$$

With Tfip for fip's time complexity, we obtain from this definition the following recurrence relations for Tfip :

$$\begin{aligned} \mathrm{Tfip}\cdot 0 &= 1 \\ \mathrm{Tfip}\cdot(n+1) &= 2 + \mathrm{Tfip}\cdot n \ , \ 0 \leqslant n \quad . \end{aligned}$$

Evaluation of fip·(n+1) requires unfolding fip once, evaluation of fip·n and unfolding of g ; hence the relation Tfip·(n+1) = 2 + Tfip·n . The solution to these relations is Tfip·n = 2 ∗ n + 1 , for n , 0 ⩽ n ; hence Tfip·n is O(n) .

□

4 Elementary programming techniques

4.0 Introduction

In this chapter we introduce a number of elementary techniques for the derivation of programs. These techniques are simple but, when applied in combination, very effective. They are elementary in the sense that they are almost always applicable. The techniques presented here are not new: probably, every programmer uses them, either consciously or subconsciously. In this chapter we try to increase their effectiveness by naming and formulating them explicitly, and by providing some heuristic guidance for their use. Moreover, we intend to show that these techniques lend themselves very well for a calculational style of programming. This style bears a strong resemblance with the, so-called, *transformational style* of program development by Burstall and Darlington [Bur][Dar0]. There is, however, one, slight but subtle, difference. We discuss this more extensively in section 4.8.

Throughout this chapter we use a single example to illustrate the use of the techniques. We call this example the *running example*. It concerns the derivation of programs for:

$$(\mathbf{S}\, i : 0 \leq i < N : X^i) \quad , \quad \text{for given natural } N \text{ and integer } X \quad .$$

Here, we impose the additional restriction that addition and multiplication are the only integer operations used. For the sake of clarity, we shall carry out all derivations in this chapter in small steps, with explicit justifications of these steps.

4.1 Replacement of constants by variables

For special values, in our running example, of the constants N and X, the problem allows simple, ad hoc, solutions. For example, we have:

$$(\mathbf{S}\, i : 0 \leqslant i < 2 : X^i) \;=\; 1 + X \quad \text{, and}$$
$$(\mathbf{S}\, i : 0 \leqslant i < N : 1^i) \;=\; N \quad .$$

The expressions $1 + X$ and N are, although solutions to instances of the same problem, not very much alike. This betrays the ad hoc ways by which these solutions have been obtained. If N and X are not so special, we may replace one or both of them by variables ranging over exactly specified *domains*. Thus, we express our recognition that the way of solving the problem should be -- actually, this is a design decision -- independent of particular properties of these constants.

Replacement of constants by variables is a form of generalisation: the value is turned into a function. This enables us to look for useful relations between the function's value in different points of its domain. If these relations take the form of, so-called, *recurrence relations*, then we may use them as a recursive definition of the function.

Because even relatively simple expressions contain many, either visible or "hidden", constants that are suitable candidates for replacement by variables, we have a methodological problem here: how do we identify the "right" candidates for replacement? Observing that we are heading for a set of recurrence relations, we propose to start as modestly as possible. We select a constant that can be replaced by a variable ranging over a domain on which *mathematical induction* is possible. Notice that mathematical induction is needed to justify the correctness of the recursive definitions to be derived. This means that the domain of the function must exhibit a partial order in such a way that it is *well-founded*. This requirement restricts the set of candidates for replacement drastically.

In our running example 0 and N are suitable candidates, whereas X is not. Generally, quantified formulae lend themselves for induction on the size of the range of quantification, provided that it is finite. Replacement of either 0 or N -- or both -- is a special case of this. If we replace N by a variable n and if we name the function thus introduced f, then we obtain the following specification for f :

$$(\mathbf{A}\, n : \mathrm{Nat}{\cdot}n : f{\cdot}n = (\mathbf{S}\, i ; 0 \leqslant i < n : X^i)) \quad .$$

In this example the choice of Nat for the range of n is rather "natural": first, n represents the *size* of the range of the quantification, and, second, with its usual ordering Nat is well-founded.

Because constant N has been replaced by variable n , it immediately follows that f·N is an expression for the value we are looking for. In order to turn it into a program, we only have to postfix this expression with a where-clause containing a definition of f satisfying the above specification.

4.2 Recurrence relations

In this section we develop a programming style based on the derivation of recurrence relations for the function for which we wish to construct a defining equation. This always involves case analysis: for the minimal elements of the domain, we strive for explicit -- i.e. nonrecursive -- expressions, whereas the relations for the other elements may be recursive. The recurrence relations thus obtained constitute a recursive definition of the function. The subsequent transformation of the recurrence relations into defining equations is mainly a matter of *encoding* these relations in the program notation. Thus, we obtain, in the program notation, a definition satisfying the specification. Provided that we carry out the derivation in a sufficiently formal way -- about which more in section 4.8 -- , the derivation of the program simultaneously constitutes its proof of correctness.

For function f , specified in the previous section, we derive a set of recurrence relations, as follows:

$f{\cdot}0$

$=$ { specification of f }

$(\mathbf{S}\, i : 0 \leqslant i < 0 : X^i)$

$=$ { empty-range rule (0 is the identity of +) }

0 ,

and:

$f\cdot(n+1)$
$=$ { specification of f }
$(\mathbf{S}\, i : 0 \leqslant i < n+1 : X^i)$
$=$ { range split }
$(\mathbf{S}\, i : 0 \leqslant i < n : X^i) + X^n$
$=$ { induction hypothesis (specification of f) }
$f\cdot n + X^n$.

We have obtained the following two relations:

$$f\cdot 0 = 0$$
$$f\cdot(n+1) = f\cdot n + X^n \ , \ 0 \leqslant n \ .$$

If we allow X^n as an expression we can encode these relations into the following program. This is a program for our original problem, in which we were interested in $f\cdot N$ only.

```
program0:      f·N |[ f·0     = 0
                    & f·(n+1) = f·n + X^n
                    ]|
```
□

If we do not allow X^n as an expression in our programs, we can choose between two strategies to eliminate it. Either we treat the subexpression as a problem in isolation by trying to derive an equivalent expression for it, or we try to derive other recurrence relations in which this subexpression does not occur. The former strategy is explored in the next sections; here, we try the latter by redoing the derivation of a relation for $f\cdot(n+1)$:

$f\cdot(n+1)$
$=$ { specification of f }
$(\mathbf{S}\, i : 0 \leqslant i < n+1 : X^i)$
$=$ { range split, but in a different way }
$X^0 + (\mathbf{S}\, i : 1 \leqslant i < n+1 : X^i)$
$=$ { 1 is the identity of $*$, dummy substitution: $i \leftarrow i+1$ }

$$
\begin{array}{ll}
 & 1 + (\mathbf{S}\, i : 0 \leq i < n : X^{i+1}) \\
= & \quad \{ X^{i+1} = X * X^i \text{ , } * \text{ distributes over } + \} \\
 & 1 + X * (\mathbf{S}\, i : 0 \leq i < n : X^i) \\
= & \quad \{ \text{induction hypothesis (specification of f)} \} \\
 & 1 + X * f{\cdot}n \quad .
\end{array}
$$

This yields:

$$f{\cdot}(n+1) = 1 + X * f{\cdot}n \quad , 0 \leq n \quad .$$

From this relation and the previous one for $f{\cdot}0$, we obtain our next program:

```
program1;      f·N |[ f·0     = 0
                   & f·(n+1) = 1 + X * f·n
                   ]|
```

□

convention 4.2.0: From here onwards, we combine range splits and subsequent dummy substitutions into a single step, without explicitly mentioning the substitution. Mostly, we are heading for a quantified formula with a range of similar form as in the formula we start with. In such cases, we have no choice as to what substitution to perform.

□

4.3 Modularisation

If, in a derivation, we encounter a subexpression -- such as X^n in our example -- that is not yet a permissible expression in the program notation, then we may decide that this subexpression forms a subproblem to be solved in isolation. To solve this subproblem we can apply the same programming techniques. In the case of our example, we may consider the parameter n in X^n as a constant that may be replaced by a natural variable. So, we may rewrite program0 as follows.

```
program2:      f·N |[ f·0     = 0
                  & f·(n+1) = f·n + g·n |[ g: (A m : 0 ≤ m : g·m = X^m ) ]|
                  ]|
```
□

Notice that in this program g's "definition" is not an admissible equation; it only serves to denote g's specification in a compact way. The correctness of f's definition only depends on this specification, not on the actual definition we supply later for g . Thus, we retain the freedom to choose *any* definition for g we like that satisfies this specification. For g , we can derive the following recurrence relations:

```
        g·0     = 1
        g·(m+1) = X * g·m   , 0 ≤ m   .
```

By plugging these relations, in encoded form, into program2 we obtain program3.

```
program3:      f·N |[ f·0     = 0
                  & f·(n+1) = f·n + g·n |[ g·0     = 1
                                         & g·(m+1) = X * g·m
                                         ]|
                  ]|
```
□

Let Tf·n and Tg·n denote the time needed to evaluate f·n and g·n respectively. Then, we have:

```
        Tf·0     = 1
        Tf·(n+1) = 1 + Tf·n + Tg·n
        Tg·0     = 1
        Tg·(n+1) = 1 + Tg·n   .
```

The solutions to these recurrence relations are:

```
        Tf·n     = n*(n+3)/2 + 1
        Tg·n     = n + 1   .
```

We conclude that program3 has quadratic time complexity.

4.4 Introduction of additional parameters

A different way to eliminate a subexpression is to replace it by a fresh name and to add this name as an additional parameter to the function for which we are developing a definition. Thus, the function is transformed into a new function with one more parameter. At first sight, this transformation only seems to make things worse: now, the subexpression must be supplied as an argument in each application of the function. Yet, the trick does sometimes work, provided that the following two conditions are met. First, in each recursive application of the function, the argument to be supplied for the new parameter can be easily expressed in terms of the function's parameters, the new parameter included. I.e, on the one hand, the introduction of the new parameter generates the obligation to supply an argument for it in each recursive application; on the other hand, the new parameter provides additional information that may be used to meet this obligation. Second, in each nonrecursive application, the other arguments must be so special that an argument for the new parameter can be constructed easily.

In order that the replacement of a subexpression E by a new parameter y does not affect the value of the expression containing E, y must represent the value of E. Therefore, we formulate the specification of the new function in such a way that its precondition implies $y = E$. More generally, it is not necessary to replace E by y; it suffices to introduce the new parameter in such a way that expression E can be replaced by a sufficiently *simple* expression in terms of y and, possibly, the other parameters.

Formally, the transformation can be described as follows. We consider function f, with the following specification, in which x represents all of f's parameters:

(0) $(\mathrm{A}x : Q{\cdot}x : R{\cdot}x{\cdot}(f{\cdot}x))$.

By introduction of a new parameter y we obtain function g with the following specification, in which P fixes the relation between the new parameter and the old ones:

(1) $(\mathbf{A}\, y,x : Q{\cdot}x \wedge P{\cdot}y{\cdot}x : R{\cdot}x{\cdot}(g{\cdot}y{\cdot}x))$.

Any application $f{\cdot}E$ may now be replaced by $g{\cdot}F{\cdot}E$, for every F satisfying $P{\cdot}F{\cdot}E$.

If we have already derived a definition for f , then we may also phrase g's specification in terms of f , as follows:

(2) $(\mathbf{A}\, y,x : Q{\cdot}x \wedge P{\cdot}y{\cdot}x : g{\cdot}y{\cdot}x = f{\cdot}x)$.

Notice that (2) is stronger than (1) , for we have: $(0) \wedge (2) \Rightarrow (1)$. We may now use the definition of f to derive a program for g . Applied in this way, the technique is a form of program transformation.

To illustrate both failure and success of this technique, we apply it in two different ways to our running example.

For the first case, we take the following recurrence relations, obtained in section 4.2, as our starting point:

$$f{\cdot}0 = 0$$
$$f{\cdot}(n+1) = f{\cdot}n + X^n \ , \ 0 \leqslant n \quad .$$

By introduction of a new parameter y to represent X^n we obtain function g with the following specification -- notice the substitution $n \leftarrow n-1$ imposed by the parameter pattern -- :

$$(\mathbf{A}\, n,y : \mathrm{Nat}{\cdot}n \wedge y = X^{n-1} : g{\cdot}y{\cdot}n = f{\cdot}n) \quad .$$

This specification is, however, awkward to use: we now have $f{\cdot}N = g{\cdot}X^{N-1}{\cdot}N$, which contains the, equally undesirable, subexpression X^{N-1} . Moreover, this expression is undefined for $N=0$. (This could be remedied by weakening the precondition for y to $n=0 \vee y=X^{n-1}$ or by replacing it by $y=X^n$.) Furthermore, for $g{\cdot}y{\cdot}(n+1)$, assuming $y=X^n$, we derive:

$$\begin{array}{ll}
& g \cdot y \cdot (n+1) \\
= & \quad \{ \text{ specification of } g,\ y = X^n \ \} \\
& f \cdot (n+1) \\
= & \quad \{ \text{ definition of } f \ \} \\
& f \cdot n + X^n \\
= & \quad \{ \ y = X^n \ \} \\
& f \cdot n + y \\
= & \quad \{ \text{ induction hypothesis (specification fo } g) \ \} \\
& g \cdot X^{n-1} \cdot n + y \\
= & \quad \{ \ y = X^n \ \} \\
& g \cdot (y/X) \cdot n + y \quad .
\end{array}$$

The formula thus obtained requires the use of division, which is meaningful for non-zero X only. Because of these complications, we reject this alternative.

Our second attempt is inspired by the symmetry of addition, on account of which we may rewrite $(\mathbf{S}\, i : 0 \leqslant i < N : X^i)$, by means of a dummy substitution $i \leftarrow N-1-i$, into $(\mathbf{S}\, i : 0 \leqslant i < N : X^{N-1-i})$. Replacement of the first occurrence of N yields the following specification for f :

$$(\mathbf{A}\, n : 0 \leqslant n \leqslant N : f \cdot n = (\mathbf{S}\, i : 0 \leqslant i < n : X^{N-1-i})) \quad .$$

In this specification, we have restricted the range of quantification to $n \leqslant N$, because we wish to consider X^{N-i-1} for natural exponents, i.e. for $i : i < N$, only.

remark 4.4.0: That it is better, in this example, to replace *both* occurrences of N by n does not concern us here.

□

By derivations similar to the ones in section 4.2, we obtain the following recurrence relations for f :

(3) $\quad f \cdot 0 = 0$

(4) $\quad f \cdot (n+1) = f \cdot n + X^{N-1-n} \quad , 0 \leqslant n < N \quad .$

By introduction of a new parameter y to represent X^{N-n} we obtain function g with the following specification:

$$(\mathbf{A}\, n,y : 0 \leq n \leq N \wedge y = X^{N-n} : g \cdot y \cdot n = f \cdot n) \quad .$$

Now, we have $f \cdot N = g \cdot 1 \cdot N$, because $X^{N-N} = 1$. Furthermore, we derive:

$g \cdot y \cdot 0$
$=$ { specification of g }
$f \cdot 0$
$=$ { (3) }
0 ,

and:

$g \cdot y \cdot (n+1)$
$=$ { specification of g }
$f \cdot (n+1)$
$=$ { (4) }
$f \cdot n + X^{N-1-n}$
$=$ { precondition of $g \cdot y \cdot (n+1)$: $y = X^{N-(n+1)}$ }
$f \cdot n + y$
$=$ { induction hypothesis (specification of g) }
$g \cdot X^{N-n} \cdot n + y$
$=$ { $X^{N-n} = X * X^{N-(n+1)}$, precondition of $g \cdot y \cdot (n+1)$ }
$g \cdot (X * y) \cdot n + y$.

Thus, we arrive at the following program.

```
program4:     g·1·N |[ g·y·0      = 0
                     & g·y·(n+1) = g·(X*y)·n + y
                     ]|
```

□

4.5 Tupling

The purpose of the technique called *tupling* is to increase a program's efficiency. Tupling can be applied to a set of function definitions, provided that these functions have the same domain and that their definitions exhibit the same pattern of recursion. In that case, we can introduce a new function, with the same domain as the functions we started with, whose value is a tuple; the elements of this tuple are the values of these functions. Due to the similarity of the patterns of recursion of the corresponding definitions, a recursive definition for the new function can be constructed easily.

Tupling can also be used to generalise a function before any program has been derived for it. This amounts to extension of the function's range. In this respect, tupling and introduction of new parameters are complementary techniques: the latter can be considered as extension of a function's domain. In most cases, however, tupling is used for program transformations directed at improvement of the program's efficiency.

We apply tupling to program3 (section 4.3) by introducing function h with specification:

$$h\cdot n = [\,f\cdot n\,,\,g\cdot n\,] \quad ,\ 0 \leqslant n \quad .$$

We have $f\cdot n = h\cdot n\cdot 0$ and $h\cdot 0 = [0,1]$; furthermore, we derive:

$h\cdot(n+1)$

$=$ { specification of h }

$[\,f\cdot(n+1)\,,\,g\cdot(n+1)\,]$

$=$ { unfolding f and g (according to program3) }

$[\,f\cdot n+g\cdot n\,,\,X*g\cdot n\,]$

$=$ { introduction of names for $f\cdot n$ and $g\cdot n$ }

$[\,a+b\,,\,X*b\,]\ |[\ [a,b] = [\,f\cdot n\,,\,g\cdot n\,]\ ]|$

$=$ { induction hypothesis for h }

$[\,a+b\,,\,X*b\,]\ |[\ [a,b] = h\cdot n\ ]|$.

Thus, we obtain the following program.

```
program5:      h·N·0 |[ h·0      = [0,1]
                     & h·(n+1) = [ a+b , X*b ] |[ [a,b] = h·n ]|
                     ]|
```

□

With Th·n denoting the time needed to evaluate h·n , we now have:

```
        Th·0      = 1  ,
        Th·(n+1) = 2 + Th·n   .
```

From these relations, it follows that Th·n = 4*n + 1 ; hence, program5 has linear time complexity.

remark 4.5.0: In example 2.8.1 we have shown that the same program can be coded in three different ways. For h·(n+1) , we could also have used as definition: h·(n+1) = c·(h·n) |[c·[a,b] = [a+b , X*b]]| .

Notice that the program for fip , in example 2.8.1, can be derived by tupling the functions fib and fib∘(+1) , where fib has been given in example 2.7.2.

□

4.6 Generalisation by abstraction

In the previous sections we have derived various programs for our running example. These programs are based on different recurrence relations; moreover, they have different properties and different time complexities. Apparently, the recurrence relations should be chosen judiciously. In order to provide some more evidence for this conclusion, we derive a program with O(log·N) time complexity. This degree of efficiency can be obtained if we can derive a recurrence relation that expresses f·n , for even n , in terms of f·(n/2) , or, equivalently, that expresses f·(2*n) in terms of f·n . Therefore, we derive, for natural n :

$f\cdot(2*n)$
= { specification of f }
$(\mathbf{S}\,i : 0 \leq i < 2*n : X^i)$
= { range split, in equal parts }
$(\mathbf{S}\,i : 0 \leq i < n : X^i) + (\mathbf{S}\,i : 0 \leq i < n : X^{n+i})$
= { $*$ distributes over $+$ (twice) }
$(1 + X^n) * (\mathbf{S}\,i : 0 \leq i < n : X^i)$
= { induction hypothesis (specification of f) (see below) }
$(1 + X^n) * f\cdot n$.

The recurrence relation thus obtained is:

$$f\cdot(2*n) = (1 + X^n) * f\cdot n \quad , \; 0 \leq n \quad .$$

This relation holds for all natural n and functions f satisfying f's specification. In order that it be useful as a recursive definition, however, the last step in the above derivation *must* be an appeal to the induction hypothesis; this requires that $n < 2*n$, i.e. n must be positive.

The relation derived above contains subexpression X^n again. Instead of eliminating it we try to avoid it by deriving another recurrence relation. The following derivation is based on the observation that a range split into equal parts can also be obtained by distinguishing even and odd numbers. The validity of this way of splitting depends on both the associativity and the symmetry of $+$:

$f\cdot(2*n)$
= { specification of f }
$(\mathbf{S}\,i : 0 \leq i < 2*n : X^i)$
= { range split, distinguishing even and odd i }
$(\mathbf{S}\,i : 0 \leq i < n : X^{2*i}) + (\mathbf{S}\,i : 0 \leq i < n : X^{2*i+1})$
= { $*$ distributes over $+$ (twice) }
$(1 + X) * (\mathbf{S}\,i : 0 \leq i < n : X^{2*i})$
= { $X^{2*i} = (X^2)^i$ }
$(1 + X) * (\mathbf{S}\,i : 0 \leq i < n : (X^2)^i)$.

Now we are stuck: the expression $(\mathbf{S}\, i : 0 \leqslant i < n : (X^2)^i)$ looks very much like, but differs from, our original expression. The only difference is that X has been replaced by X^2 . Therefore, we may consider the two expressions as instances of a single, more general expression. This amounts to a posteriori replacement of constants by variables. In our example, both expressions are instances of $(\mathbf{S}\, i : 0 \leqslant i < n : x^i)$, for integer x . Variable x is a new parameter of the function; if we call the function thus introduced g , then its specification is:

$$(\mathbf{A}\, x,n : \mathrm{Int}{\cdot}x \wedge \mathrm{Nat}{\cdot}n : g{\cdot}x{\cdot}n = (\mathbf{S}\, i : 0 \leqslant i < n : x^i)) \quad .$$

We call the technique used here *generalisation by abstraction*. Notice that the same effect could have been achieved by replacement of the *relevant* constants by variables right from the start. It is, however, not clear a priori what the relevant constants are. The crux is that *two* -- or more, for that matter -- slightly different expressions provide more information than one. Notice that in this way the generalisation needed is, to a large extent, discovered by calculation. Moreover, provided that the calculations leading to this discovery have been carried out sufficiently carefully, these calculations usually need not be redone for the generalisation: if the derivation does not depend on properties lost in the generalisation, it still pertains, mutatis mutandis, to the generalised case. Thus, this way of working is not necessarily inefficient.

In our example, no properties, except being an integer, of X have been used. Hence, without further formal labour, we obtain the following recurrence relations for g from the corresponding relations for f :

$$\begin{array}{lll}
g{\cdot}x{\cdot}0 & = 0 & \\
g{\cdot}x{\cdot}(n+1) & = g{\cdot}x{\cdot}n + x^n & ,\ 0 \leqslant n \\
g{\cdot}x{\cdot}(n+1) & = 1 + x * g{\cdot}x{\cdot}n & ,\ 0 \leqslant n \\
g{\cdot}x{\cdot}(2*n) & = (1 + x^n) * g{\cdot}x{\cdot}n & ,\ 1 \leqslant n \\
g{\cdot}x{\cdot}(2*n) & = (1 + x) * g{\cdot}x^2{\cdot}n & ,\ 1 \leqslant n \quad .
\end{array}$$

Using the first, the third, and the fifth relation, we construct the following program, in which peven·n and podd·n are used as abbreviations of $1 \leqslant n \wedge n \,\mathbf{mod}\, 2 = 0$ and $1 \leqslant n \wedge n \,\mathbf{mod}\, 2 = 1$. Its time complexity is $O(\log{\cdot}N)$.

```
program6:      g·X·N |[ g·x·0 = 0
                     & g·x·n = podd·n   → 1 + x * g·x·(n-1)
                                [] peven·n → (1 + x) * g·(x*x)·(n div 2)
                     ]|
```
□

4.7 Linear and tail recursion

In this section we discuss a common pattern of recursion, called *linear recursion*, and a special case thereof, called *tail recursion*. Tail recursion is of interest for, at least, two reasons. First, tail recursive definitions tend to require less storage space for their evaluation than generally recursive definitions [Pey]. Second, tail recursive definitions can be translated into repetitions in a sequential-program notation in a straightforward way. This is important when we use functional programming to design programs to be implemented in a sequential-program notation.

The transformation of a recursive definition into a tail recursive one is an example of a program transformation. In this area, much research has already been done [Bur][Dar0][Hen]. The tail recursion theorem derived in this section is not new, but its derivation is a nice application of generalisation by abstraction: by means of this technique, the theorem emerges in a straightforward way.

Although we use functional-program notation to define functions, the discussion in this section pertains to functions in general; i.e, its validity is not restricted to functions defined in our program notation.

definition 4.7.0 (linear recursion): A recursive definition of a function is called *linearly recursive* if each unfolding of an application of that function generates at most one recursive application of the function.

□

The following definition of function F may be considered as the prototype of a linearly recursive definition:

$$
\begin{array}{lll}
(0) & F{\cdot}x = (& \neg b{\cdot}x \rightarrow f{\cdot}x \\
 & & [\!]\ b{\cdot}x \rightarrow h{\cdot}x \oplus F{\cdot}(g{\cdot}x) \\
 & &) \quad .
\end{array}
$$

In the following discussion, U, V, and W are sets, $\leqslant$ denotes a partial order on U , and b, f, g, h, and $\oplus$ are functions with the following properties. Properties (1) through (5) state the types of these functions, whereas properties (6) and (7) enable us to use mathematical induction on U . From these properties it follows, by mathematical induction on U , that F has type $U \rightarrow W$.

(1)	$(\mathbf{A}\, x : U{\cdot}x : Bool{\cdot}(b{\cdot}x))$	("b has type $U \rightarrow Bool$")
(2)	$(\mathbf{A}\, x : U{\cdot}x \wedge \neg b{\cdot}x : W{\cdot}(f{\cdot}x))$	("f has type $U \cap \neg b \rightarrow W$")
(3)	$(\mathbf{A}\, x : U{\cdot}x \wedge\ b{\cdot}x : U{\cdot}(g{\cdot}x))$	("g has type $U \cap b \rightarrow U$")
(4)	$(\mathbf{A}\, x : U{\cdot}x \wedge\ b{\cdot}x : V{\cdot}(h{\cdot}x))$	("h has type $U \cap b\,) \rightarrow V$")
(5)	$(\mathbf{A}\, y,z : V{\cdot}y \wedge W{\cdot}z : W{\cdot}(y \oplus z))$	("$\oplus$ has type $V \times W \rightarrow W$")
(6)	$(U,\leqslant)$ is well-founded	
(7)	$(\mathbf{A}\, x : U{\cdot}x \wedge\ b{\cdot}x : g{\cdot}x < x)$	

Strictly speaking, function h in definition (0) is superfluous: define $\otimes$ by $x \otimes z = h{\cdot}x \oplus z$ and replace $\oplus$ by $\otimes$. The redundancy provided by the presence of h , however, leaves us more freedom in the choice of $\oplus$.

The purpose of this discussion is to explore what we can derive about F by application of generalisation by abstraction. We restrict our attention to the special case $V = W$; then, $\oplus$ is a binary operator on $W \times W$. We observe that $F{\cdot}x$ and $h{\cdot}x \oplus F{\cdot}(g{\cdot}x)$ are instances of the more general expression $y \oplus F{\cdot}x$, provided that $\oplus$ has a left identity, say, e ; then, $F{\cdot}x = e \oplus F{\cdot}x$. Therefore, we assume (8) , with:

(8) e is a left identity of $\oplus$.

We now introduce function G with the following specification:

$$(\mathbf{A}\, y,x : W{\cdot}y \wedge U{\cdot}x : G{\cdot}y{\cdot}x = y \oplus F{\cdot}x) \quad .$$

On account of this specification and (8) , we have $(\mathbf{A}\, x : U{\cdot}x : F{\cdot}x = G{\cdot}e{\cdot}x)$.

Using mathematical induction on U , we derive for G :

case $\neg b \cdot x$:

$$
\begin{aligned}
& G \cdot y \cdot x \\
= & \quad \{ \text{ specification of } G \ \} \\
& y \oplus F \cdot x \\
= & \quad \{ \text{ unfolding } F \text{, using } \neg b \cdot x \ \} \\
& y \oplus f \cdot x \quad .
\end{aligned}
$$

case $b \cdot x$:

$$
\begin{aligned}
& G \cdot y \cdot x \\
= & \quad \{ \text{ specification of } G \ \} \\
& y \oplus F \cdot x \\
= & \quad \{ \text{ unfolding } F \text{, using } b \cdot x \ \} \\
& y \oplus (h \cdot x \oplus F \cdot (g \cdot x)) \\
= & \quad \{ \text{ assume } (9) \text{, see below } \} \\
& (y \oplus h \cdot x) \oplus F \cdot (g \cdot x) \\
= & \quad \{ \ (7) \wedge b \cdot x \Rightarrow g \cdot x < x \text{ : induction hypothesis for } G \ \} \\
& G \cdot (y \oplus h \cdot x) \cdot (g \cdot x) \quad .
\end{aligned}
$$

Due to (1) , the case analysis $\neg b \cdot x \vee b \cdot x$ is exhaustive. In this derivation we have assumed (9) , with:

(9) $\oplus$ is associative .

Putting the pieces together, we obtain the following recursive definition for G :

(10) $G \cdot y \cdot x = (\neg b \cdot x \ \rightarrow \ y \oplus f \cdot x$
$\qquad\qquad\qquad [] \ b \cdot x \ \rightarrow \ G \cdot (y \oplus h \cdot x) \cdot (g \cdot x)$
$\qquad\qquad\qquad)$.

In fact, we now have derived the following theorem.

theorem 4.7.1 (tail recursion theorem): For associative $\oplus$ with left identity e , and for F and G defined by (0) and (10) , we have:

$$(\mathbf{A}\, x : U{\cdot}x : F{\cdot}x = G{\cdot}e{\cdot}x) \quad .$$

□

The definition of G is a special kind of a linearly recursive definition, called a *tail recursive* definition. Generally, the definition of function H is tail recursive if it has the following form:

```
H·x = (¬b·x → f·x
      ▯ b·x → H·(g·x)
      )    .
```

Tail recursive definitions can be easily translated into sequential programs; here is a sequential program for the computation of $H{\cdot}X$, for given X ; in it, we use a variable x whose role resembles the role of H's parameter in the above definition:

```
 x := X  { invariant: H·X = H·x }
;do  b·x → { H·x = H·(g·x) }  x := g·x
 od { H·X = H·x ∧ ¬b·x , hence: }
{ H·X = f·x }    .
```

By instantiation of this program, we obtain a sequential program for the computation of $F{\cdot}X$; this amounts to coding (10) as a sequential program for the computation of $G{\cdot}e{\cdot}X$; hence, its invariant would be $G{\cdot}e{\cdot}X = G{\cdot}y{\cdot}x$. By application of G's specification, we can reformulate this invariant directly in terms of F ; the invariant thus obtained is usually called a *tail invariant*. This gives the following program:

```
 y,x := e,X  { invariant: F·X = y ⊕ F·x }
;do  b·x → y,x := y ⊕ h·x , g·x
 od
{ F·X = y ⊕ f·x }    .
```

Depending on the actual structure of the definition of F -- in our case: (0) -- the above theme allows many variations. Therefore, it is not so much the tail recursion theorem itself but its derivation, based on generalisation by abstraction, that is important. To illustrate this, we use the same approach to derive a sequential program, with $O(\log{\cdot}N)$ time complexity, for our running example. Although slightly more complicated, the following derivation essentially is the same as the derivation of the tail recursion theorem. It depends on additional algebraic properties such as that 0, apart from being an identity of $+$, also is a zero of $*$, that $*$ is associative, et cetera.

From section 4.6 we recall that $g{\cdot}X{\cdot}N$ provides a solution for the problem, and we recall the following recurrence relations for g:

$$
\begin{array}{lll}
g{\cdot}x{\cdot}0 & = 0 & \\
g{\cdot}x{\cdot}(n+1) & = g{\cdot}x{\cdot}n + x^n & ,\ 0 \leqslant n \\
g{\cdot}x{\cdot}(n+1) & = 1 + x * g{\cdot}x{\cdot}n & ,\ 0 \leqslant n \\
g{\cdot}x{\cdot}(2*n) & = (1 + x^n) * g{\cdot}x{\cdot}n & ,\ 1 \leqslant n \\
g{\cdot}x{\cdot}(2*n) & = (1 + x) * g{\cdot}x^2{\cdot}n & ,\ 1 \leqslant n \quad .
\end{array}
$$

These relations provide several possibilities for generalisation by abstraction, giving rise to formulae such as $z + y * g{\cdot}x{\cdot}n$, or $z + (1+y) * g{\cdot}x{\cdot}n$, or $y * g{\cdot}x{\cdot}n + z * x^n$. Selecting the first one, we introduce function h with the following specification, as a result of which we have $g{\cdot}x{\cdot}n = h{\cdot}0{\cdot}1{\cdot}x{\cdot}n$:

$$(\mathbf{A}\, z,y,x,n :: h{\cdot}z{\cdot}y{\cdot}x{\cdot}n = z + y * g{\cdot}x{\cdot}n \,) \quad .$$

Using the first, third, and last of the recurrence relations for g, we can derive the following relations for h:

$$
\begin{array}{lll}
h{\cdot}z{\cdot}y{\cdot}x{\cdot}0 & = z & \\
h{\cdot}z{\cdot}y{\cdot}x{\cdot}(n+1) & = h{\cdot}(z+y){\cdot}(y*x){\cdot}x{\cdot}n & ,\ 0 \leqslant n \\
h{\cdot}z{\cdot}y{\cdot}x{\cdot}(2*n) & = h{\cdot}z{\cdot}(y*(1+x)){\cdot}(x*x){\cdot}n & ,\ 1 \leqslant n \quad .
\end{array}
$$

Encoding these relations in the program notation yields the following program.

```
program7:       h·0·1·X·N |[ h·z·y·x·0 = z
                          & h·z·y·x·n = ( podd·n  → h·(z+y)·(y*x)·x·(n-1)
                                        [] peven·n → h·z·(y*(1+x))·(x*x)·(n div 2)
                                        )
                          ]|
```
□

The definition of h , in this program, is tail recursive. From this program, the following sequential program can be derived. As before, by instantiation of the standard invariant h·0·1·X·N = h·z·y·x·n , the invariant of the program's repetition can be formulated in terms of function g .

```
program8:       z,y,x,n := 0,1,X,N  { invariant: g·X·N = z + y*g·x·n }
               ;do podd·n   → z,y,n := z+y , y*x , n-1
                 [] peven·n → y,x,n := y*(1+x) , x*x , n div 2
                od
               { z = (S i : 0 ≤ i < N : X^i) }
```
□

4.8 Mainly on presentation

In this section we discuss the way in which we present derivations of programs. Moreover, we compare our approach with the, so-called, Burstall/ Darlington [Bur][Dar0] style of program development.

In order that program development by calculation be practically usable, the process of formula manipulation should be sufficiently efficient. Particularly, we wish to avoid duplication of formal labour, and we wish to avoid copying, over and over again, large formulae that remain constant during the derivation. The desire to avoid duplication of work brings about that we do not want to give separate, a posteriori, proofs of correctness of our programs. We see to it that the derivations of our programs simultaneously are their proofs of correctness. Ideally, the derivation is presented in such a way that it is no longer than the corresponding a posteriori proof would have been. This approach leaves us the freedom to decide how much heuristic explanation will be included in the presentation; the basic pattern of the presentation, however, will be largely independent of this decision. Finally, whether or not heuristic explanation is provided, the presentation should be such that it is clear what steps in the derivation are design decisions and what steps are mere simplifications.

By means of a very simple example we illustrate a few different ways to present derivations. The example is the derivation of a recursive definition for function f , having type $Nat \rightarrow Nat$, satisfying:

(0) $(\mathbf{A}\, i : 0 \leq i : f{\cdot}i = i^2)$.

The example is a little bit insipid in the sense that (0) can be strenghtened immediately to $(\mathbf{A}\, i :: f{\cdot}i = i * i)$, which is an admissible equation, but this need not bother us here.

A derivation that clearly shows the correctness of the design is:

$$
\begin{array}{ll}
 & (\mathbf{A}\, i : 0 \leq i : f{\cdot}i = i^2) \\
\Leftarrow & \quad \{ \text{ mathematical induction on Nat } \} \\
 & f{\cdot}0 = 0 \wedge (\mathbf{A}\, i : 0 \leq i : f{\cdot}i = i^2 \Rightarrow f{\cdot}(i+1) = (i+1)^2) \\
\Leftarrow & \quad \{ \text{ calculus } \} \\
 & f{\cdot}0 = 0 \wedge (\mathbf{A}\, i : 0 \leq i : f{\cdot}(i+1) = f{\cdot}i + 2*i + 1) \quad .
\end{array}
$$

The formula thus obtained can be considered as an admissible equation for f ; it can be encoded in the program notation as follows:

(1) $f{\cdot}0 = 0 \;\&\; f{\cdot}(i+1) = f{\cdot}i + 2*i + 1$.

The correctness of this definition follows from the fact that, in the above derivation, the specification has been strengthened only. Observing that equality is symmetric, we may rewrite the equation $f{\cdot}(i+1) = f{\cdot}i + 2*i + 1$ to the equivalent $f{\cdot}i = f{\cdot}(i+1) - 2*i - 1$; hence, what is wrong with the following definition?

(2) $f{\cdot}0 = 0 \;\&\; f{\cdot}i = f{\cdot}(i+1) - 2*i - 1$.

To analyse this definition, we observe that it is, according to definitions 2.7.1 and 2.7.0, an abbreviation of:

$$f\cdot i \;=\; (\, i=0 \to 0 \;[]\; i\geq 0 \to f\cdot(i+1) - 2*i - 1 \,) \quad .$$

With the proof rule for guarded selections, the strongest proposition we can prove about this f is: $(f\cdot 0 = 0 \lor f\cdot 0 = f\cdot 1 - 1) \land (\mathbf{A}\, i : 1\leq i : f\cdot i = f\cdot(i+1) - 2*i - 1)$, which is weaker than what we need, viz. $f\cdot 0 = 0 \land (\mathbf{A}\, i : 0\leq i : f\cdot i = f\cdot(i+1) - 2*i - 1)$. Hence, (2) must be rejected. The moral of this story is, of course, that the arguments in recursive applications of a function must be less than the parameters of the function; here, "less than" refers to the partial order imposed onto the function's domain to justify the use of mathematical induction.

The above derivation is rather short, because the example is so simple. In more realistic examples, we prefer to deal with the two cases -- "base" and "step", so to speak -- , arising from the use of mathematical induction, separately. Moreover, we do not want to carry around the (constant!) induction hypothesis. The proof obligation is to show that the definition derived satisfies the specification of the function. The induction hypothesis *always* is that the definition satisfies the specification for all arguments *less than* the argument under investigation. That is, given the function's specification and given the partial order used on the function's domain, it is clear what the induction hypothesis should be. Therefore, we can afford not to write down the induction hypothesis explicitly.

For our simple example, these observations lead to a derivation of the following form. Starting from (0) , we use mathematical induction on Nat , and we derive -- the base -- :

$$f\cdot 0 = 0$$
$$\Leftarrow \quad \{ \text{ choose as definition: } f\cdot 0 = 0 \;\}$$
$$\text{true} \quad .$$

The step in this derivation looks rather stupid, but it really embodies a design decision: for instance, we also could have chosen $f\cdot x = x$ or $f\cdot x = x*x$. In view of the simplicity of (0) , the latter one is even a better proposal. Yet, the choice $f\cdot 0 = 0$ is the least committing one, because it restricts our freedom to define f in *other* points of its domain as little as possible. In this respect, we have no real freedom here. In practice, we omit the last step and record this design decision separately, for instance by writing down (a fragment of) a program text.

Next, we derive, for natural i -- the step -- :

$f{\cdot}(i+1) = (i+1)^2$

$=$ { algebra }

$f{\cdot}(i+1) = i \quad + 2*i + 1$

$\Leftarrow$ { induction hypothesis for f }

$f{\cdot}(i+1) = f{\cdot}i + 2*i + 1$.

The formula thus obtained can be used as a definition; combination with the result obtained for the base case then yields definition (1) .

The above derivations have been carried out in the domain of predicate calculus. This is particularly useful if the specification takes the form of an implicit equation for the value specified by it. When, as is often the case with specifications of functions, the specification provides an explicit formula for the function's values, then the derivation can also be carried out in the domain of values. This avoids copying the constant left-hand side of the equation. For our example, such a derivation takes the following form.

$f{\cdot}0$

$=$ { specification of f }

0 .

Hence, we choose $f{\cdot}0 = 0$ as a definition. Furthermore, for natural i we derive:

$f{\cdot}(i+1)$

$=$ { specification of f }

$(i+1)^2$

$=$ { algebra }

$i^2 + 2*i + 1$

$=$ { induction hypothesis for f }

$f{\cdot}i + 2*i + 1$.

Hence, for this case, we choose $f{\cdot}(i+1) = f{\cdot}i + 2*i + 1$ as a definition. Again, by combination of these results we obtain definition (1) .

In the Burstall/Darlington style of program development, our small example problem would be solved as follows. This style is *transformational*, which requires that the specification already is a, possibly very inefficient, program. In our case, (0) may be encoded as a defining equation as follows.

(3) $\quad f{\cdot}i = (\, 0 \leq i \rightarrow i * i \,)$.

We now derive:

$f{\cdot}0$

$=\quad$ { (3) , i.e: unfolding f }

0 ,

and:

$f{\cdot}(i+1)$

$=\quad$ { (3) , i.e: unfolding f }

$(i+1)*(i+1)$

$=\quad$ { algebra }

$i*i + 2*i + 1$

$=\quad$ { (3) , i.e: folding f }

$f{\cdot}i + 2*i + 1$.

Thus, we may add the following definitions to our set of definitions for f :

(4) $\quad f{\cdot}0 = 0 \;\;\&\;\; f{\cdot}(i+1) = f{\cdot}i + 2*i + 1$.

Notice, however, that we actually have derived $(3) \Rightarrow (4)$ only: by the above derivation, we have weakened instead of strengthened the specification. In order to conclude that (4) indeed is a correct definition satisfying (3) , there is a remaining proof obligation. Burstall and Darlington call this the obligation to prove *termination*, and state that the programs thus obtained are *partially correct* only. Apart from this, the derivation given here corresponds exactly to the last derivation given above. The only flaw in the derivation given here is the step with hint "folding f" : if we had included an appeal to the induction hypothesis, then we would have obtained a correct program *without* further proof obligations. In this respect, we wish to stress that the use of induction, right from the start, has heuristic value too: it restricts the freedom we have when we apply folding. Phrased differently, in the Burstall/Darlington style the programmer needs some clairvoyance in the application of folding, in order to guarantee that the remaining proof obligation can be met. It is, for instance, possible to derive the wrong definition (2) in this way, but for this definition termination cannot be proved.

4.9 Discussion

The technique of generalisation by abstraction is important for two reasons. First, it reduces the amount of foresight needed by the programmer considerably. It is easier to discover what constants are candidates for replacement by analysing the differences between a number of similar expressions than to guess for a single expression what its relevant aspects are. Therefore, it seems to be a wise strategy to replace only one constant by a variable, on whose domain the induction will be based; by the subsequent derivation of recurrence relations we try to discover what else is needed. Second, due to its general applicability, the technique opens up the way to a large class of programs. For our running example, we have shown only a few of these programs; many more variations are possible.

The reverse side of this coin is that programming in this way cannot be a blind, mechanical activity. Even for a relatively simple problem as our running example, a large number of recurrence relations exist that can be used in many ways. This being so, we prefer techniques that identify as many interesting solutions as possible in an early stage of the design process, and, of course, without causing an explosion of formal labour.

The programs we have derived for our running example are not completely equivalent, in the sense that their correctness depends on different algebraic properties of 0, 1, +, and $*$. By careful identification of these properties we discover for what values X the programs may be used. We have, for instance, not really used that X is an integer; all programs derived in this chapter are correct when X is element of a *ring* with multiplicative identity. X may, for instance, be a matrix.

5 Theory of lists

5.0 Introduction

In this chapter we define *lists* and we develop some theory for their use. Lists form a special kind of functions either on initial segments of Nat or on Nat itself. The former are called *finite lists*, whereas the latter are called *infinite lists*. The values of a list in various points of its domain are called *elements* of the list. Lists differ from other functions on (initial segments of) Nat in the way they are implemented. The difference is that each element of a list is assumed to be evaluated at most once; the result of this evaluation is stored, so that it can be used more than once without further evaluation. On the other hand, multiple applications, to the same argument, of a function that is not a list will generally give rise to multiple evaluations of that same application. Thus, lists can be used to save computation time, at the expense of storage space; in this respect, lists are the direct counterpart of *arrays* in sequential-program notations.

From the above one might conclude that infinite lists require an infinite amount of storage space for their representation and that they are, therefore, not representable in any computer. Notice, however, that, at any moment during a computation, at most finitely many list elements have been computed. Hence, the computed part of a list is always finitely representable. Phrased differently, complete evaluation of an infinite list takes an infinite amount of time and until that moment a finite amount of storage space suffices. In this respect, infinite lists do not differ from other expressions.

In our notation there is no formal difference between lists and tuples, as introduced in chapter 2: tuples simply *are* finite lists. The difference between tuples and lists lies in the way they are used; usually, all elements of a list are assumed to have the same type, whereas the elements of a tuple may have different types. Because, in our formalism, there is no syntactic notion of type, we do not need different notations for tuples and for lists.

5.1 Primitives for list construction

In this section we introduce a number of constants and functions, called the *list primitives*, in terms of which lists will be defined. All properties and theorems about lists formulated in the following sections can be proved by means of the properties of these primitives.

definition 5.1.0 (syntax of the list primitives): The list primitives are:

the constant []	("the empty list", or "empty")
the function ise	("is empty")
the binary operator ;	("cons")
the binary operator $\uparrow$	("take")
the binary operator $\downarrow$	("drop")

The binary operators are used in infix notation. ; binds weaker than $\uparrow$ and $\downarrow$; furthermore, ; is right-binding, whereas $\uparrow$ and $\downarrow$ are left-binding. According to our general convention, $\cdot$ and $\circ$ bind stronger than ; , $\uparrow$, and $\downarrow$. The binding power of these operators relative to other operators is irrelevant; we shall always use parentheses when necessary.

□

postulate 5.1.1 (semantics of the list primitives): The list primitives satisfy the following relations; they hold for all x, y, i , $\Omega\cdot x \wedge \Omega\cdot y \wedge \mathrm{Nat}\cdot i$:

$$\begin{aligned}
\mathrm{ise}\cdot[\,] &\equiv \mathrm{true}\\
\mathrm{ise}\cdot(x\,;y) &\equiv \mathrm{false}\\
(x\,;y)\cdot 0 &= x\\
(x\,;y)\cdot(i+1) &= y\cdot i\\
x\uparrow 0 &= [\,]\\
[\,]\uparrow i &= [\,]\\
(x\,;y)\uparrow(i+1) &= x\,;y\uparrow i\\
x\downarrow 0 &= x\\
[\,]\downarrow i &= [\,]\\
(x\,;y)\downarrow(i+1) &= y\downarrow i
\end{aligned}$$

□

remark 5.1.2: Actually, the set of list primitives is redundant: instead of $\uparrow$ and $\downarrow$, with the above properties, we could have introduced $(\downarrow 1)$ only, with properties $[]\downarrow 1 = []$ and $(x;y)\downarrow 1 = y$. In terms of these properties, the operators $\uparrow$ and $\downarrow$ can then be defined. For the sake of simplicity and symmetry, however, we have decided to introduce $\uparrow$ and $\downarrow$ right away. In the traditional literature on functional programming, two functions hd ("head") and tl ("tail") are used, satisfying $hd\cdot(x;y) = x$ and $tl\cdot(x;y) = y$; notice that hd and tl correspond to $(\cdot 0)$ and $(\downarrow 1)$.

□

remark 5.1.3: As is the case with other constructs in our program notation, the above postulate does not specify the values of the list primitives completely. Yet, this postulate provides all information about the list primitives we need, and nothing else.

□

The following properties follow immediately from the above definition.

property 5.1.4: for all u, v, x, y :

$(x;y) \neq []$

$(x;y)\downarrow 1 = y$

$(u;v) = (x;y) \equiv u = x \wedge v = y$

□

convention 5.1.5: By abuse of notation, we write $x = []$ instead of ise·x , and $x \neq []$ instead of $\neg$ise·x . Notice that, according to postulate 5.1.1, if $x = [] \vee (E\, y,z :: x = y;z)$ then the expression $x = []$ has a boolean value. For all other values of x the value of the expression $x = []$ must be considered as undefined, i.e. completely unspecified.

□

5.2 Listoids, finite lists, and infinite lists

In this section we give a formal definition of, both finite and infinite, lists and a few elementary properties. Usually, all elements of a list have the same type. Here, we use A to denote the element type of the lists defined. Notice that A may be Ω , in which case no restrictions are imposed on the type of the elements. Throughout this section variables a, b denote values of type A , whereas variables x, y denote values from Ω .

definition 5.2.0 (finite lists): The *finite lists over A of length i* , for natural i , are the elements of the set $L_i(A)$; these sets are defined recursively by:

$$\begin{aligned} L_0(A)\cdot x &\equiv x = [\,] \\ L_{i+1}(A)\cdot x &\equiv (\mathbf{E}\, b,y : A\cdot b \wedge L_i(A)\cdot y : x = b\,;y) \quad . \end{aligned}$$

Informally, the elements of $L_i(A)$ are of the form $a_0\,;a_1\,;\ldots;a_{i-1}\,;[\,]$, for values a_j $(0\leqslant j<i)$, $A\cdot a_j$. These values are called the *elements* of the list. The set $L_*(A)$, of *finite lists over A* , is the union of the sets $L_i(A)$, i.e:

$$L_*(A)\cdot x \equiv (\mathbf{E}\, i : 0\leqslant i : L_i(A)\cdot x) \quad .$$

□

The finite lists have been defined by means of the auxiliary notion of finite lists of length i . In a similar way, we need an auxiliary notion, namely the *listoids of order i* , for the definition of infinite lists.

definition 5.2.1 (listoids and infinite lists): The *listoids over A of order i* , for natural i , are the elements of the set $P_i(A)$; these sets are defined recursively by:

$$\begin{aligned} P_0(A)\cdot x &\equiv \text{true} \\ P_{i+1}(A)\cdot x &\equiv (\mathbf{E}\, b,y : A\cdot b \wedge P_i(A)\cdot y : x = b\,;y) \quad . \end{aligned}$$

Informally, the elements of $P_i(A)$ are of the form $a_0\,;a_1\,;\ldots;a_{i-1}\,;x$, for values a_j $(0\leqslant j<i)$, $A\cdot a_j$, and for any x . Again, the values a_j are called *elements* of the listoid. The set $L_\infty(A)$, of *infinite lists over A*, is the intersection of the sets $P_i(A)$, i.e:

$$L_\infty(A)\cdot x \equiv (\mathbf{A}\, i : 0\leqslant i : P_i(A)\cdot x) \quad .$$

□

definition 5.2.2 (lists): A *list over A* is either a finite or an infinite list over A. The union of the sets $L_*(A)$ and $L_\infty(A)$ is called $L(A)$, i.e:

$$L(A)\cdot x \quad \equiv \quad L_*(A)\cdot x \vee L_\infty(A)\cdot x \quad .$$

□

For finite lists we introduce an abbreviation. This notation is the same as the notation for tuples defined in chapter 2: tuples are finite lists.

definition 5.2.3 (notation for finite lists): For natural n and values $a_i (0 \leqslant i < n)$:

$$[\, a_0 , \dots , a_{n-1} \,] = a_0 ; \dots ; a_{n-1} ; [] \quad .$$

□

Whenever it is clear from the context what the element type of the lists we are discussing is, or when $A = \Omega$, we omit the type indication; so, we write L_* instead of $L_*(A)$, etcetera. In that case, we also speak of (finite or infinite) *lists* instead of *lists over A* . In section 5.3 we show that the proof that a value is a list can be separated completely from the proof that its elements satisfy certain properties.

We conclude this section with a selection of simple properties of the sets defined above. The proofs of these properties are neither very difficult nor very interesting; we, therefore, omit them.

property 5.2.4: For natural i, j :

$P_{i+1} \subseteq P_i$, hence:

$L_\infty \cdot x \equiv (\mathbf{A}\, i : j \leqslant i : P_i \cdot x)$

$L_i \subseteq P_i$

$L_\infty \subseteq P_i$

$L_i \cap L_j = \emptyset \quad \vee \quad i = j$

$L_i \cap P_{i+1} = \emptyset$, hence: $L_i \cap L_\infty = \emptyset$, hence:

$L_* \cap L_\infty = \emptyset$

□

property 5.2.5: For natural i, j :

$(\uparrow i)$ has type $P_j \to L_i$, for i,j : $i \leq j$
$(\uparrow i)$ has type $L_j \to L_{i \mathbf{min} j}$
$(\uparrow i)$ has type $L_\infty \to L_i$
$(\downarrow i)$ has type $P_j \to P_{(j-i)\mathbf{max}0}$
$(\downarrow i)$ has type $L_j \to L_{(j-i)\mathbf{max}0}$
$(\downarrow i)$ has type $L_\infty \to L_\infty$

□

property 5.2.6: For natural i , element a (i.e: A·a), and value x :

$$
\begin{aligned}
L_{i+1}\cdot(a\,;x) &\equiv L_i\cdot x\\
L_\star\cdot(a\,;x) &\equiv L_\star\cdot x\\
P_{i+1}\cdot(a\,;x) &\equiv P_i\cdot x\\
L_\infty\cdot(a\,;x) &\equiv L_\infty\cdot x\\
L_\star\cdot x &\equiv x=[\,] \vee (\mathbf{E}\, b,y : L_\star\cdot y : x=b\,;y)\\
L\cdot x &\equiv x=[\,] \vee (\mathbf{E}\, b,y : L\cdot y : x=b\,;y)\\
L_\infty\cdot x &\equiv (\mathbf{E}\, b,y : L_\infty\cdot y : x=b\,;y)
\end{aligned}
$$

□

corollary 5.2.7: $(\mathbf{A}\, x : L\cdot x : x=[\,] \vee x=x\cdot 0\,; x\downarrow 1)$.

□

corollary 5.2.8: For predicate R we have:

$$
\begin{aligned}
(\mathbf{A}\, x : L_\star\cdot x : R\cdot x) &\equiv R\cdot[\,] \wedge (\mathbf{A}\, b,y : L_\star\cdot y : R\cdot(b\,;y))\\
(\mathbf{A}\, x : L_\infty\cdot x : R\cdot x) &\equiv (\mathbf{A}\, b,y : L_\infty\cdot y : R\cdot(b\,;y))\\
(\mathbf{A}\, x : L\cdot x : R\cdot x) &\equiv R\cdot[\,] \wedge (\mathbf{A}\, b,y : L\cdot y : R\cdot(b\,;y))
\end{aligned}
$$

□

remark 5.2.9: Property 5.2.6 shows that both $L_\star$ and L are solutions of the equation, with unknown P : $(\mathbf{A}\, x :: P\cdot x \equiv x=[\,] \vee (\mathbf{E}\, b,y : P\cdot y : x=b\,;y))$. Actually, it can be shown that $L_\star$ is the *smallest* and L is the *greatest* solution of this equation. Similarly, L_∞ is the greatest solution of the equation, with unknown P : $(\mathbf{A}\, x :: P\cdot x \equiv (\mathbf{E}\, b,y : P\cdot y : x=b\,;y))$.

□

5.3 The length of a list

We use the unary operator # ("length" or "size") to denote the *length* of a list. It has type $L \to (\mathrm{Nat} \cup \{\infty\})$, where ∞ ("infinity") is used to denote the length of infinite lists; we postulate that $(\mathrm{A}\, i : \mathrm{Nat}\cdot i : i < \infty)$. We use ∞ only in discussions about lists. In programs, we apply # to finite lists only. Thus, value ∞ need not be computable.

postulate 5.3.0: Value ∞ satisfies:

$$(\mathrm{A}\, i : \mathrm{Nat}\cdot i : i < \infty) \quad .$$

□

definition 5.3.1: Operator # satisfies:

$$(\mathrm{A}\, i,s : \mathrm{Nat}\cdot i \wedge L_i\cdot s : \#s = i)$$
$$(\mathrm{A}\, s : L_\infty\cdot s : \#s = \infty) \quad .$$

□

From these definitions, the definition of lists, and property 5.2.6, it follows that # has the following properties.

property 5.3.2: For any value a , finite list s , and infinite list x :

$$\#[\,] \quad = 0$$
$$\#(a\,;s) = 1 + \#s \;, \quad \text{hence:} \quad \#s < \#(a\,;s)$$
$$\#(a\,;x) = \#x$$

For any (finite or infinite) list x :

$$\#x = 0 \quad \equiv \quad x = [\,]$$
$$\#x > 0 \quad \equiv \quad (\mathrm{E}\, b,y : L\cdot y : x = b\,;y)$$

□

By means of ∞ we can identify the elements of a list without distinguishing finite and infinite lists: the elements of list x are $x\cdot i$, $0 \leqslant i < \#x$. For example, the following property expresses that lists over some type A are lists over Ω whose elements have type A . Hence, that x is a list over A can be proved by showing, first, that x is a list (over Ω), and, second, that all elements of x have type A . For the latter, we need not know that x is a list: x may be treated as any other function on (an initial segment of) Nat .

property 5.3.3: For type A , and any x :

$$L(\mathrm{A})\cdot x \quad \equiv \quad L(\Omega)\cdot x \wedge (\mathrm{A}\, i : 0 \leqslant i < \#x : \mathrm{A}\cdot(x\cdot i)) \quad .$$

□

property 5.3.4: For type A , and any x :

$$L(A)\cdot x \quad \Rightarrow \quad \text{"x has type } \{ i \mid 0 \leq i < \#x \} \rightarrow A \text{ "}$$

□

5.4 Theorems for finite lists

Because $\#$ is a function of type $L_\star \rightarrow \text{Nat}$, properties of finite lists may be proved by mathematical induction on the value of $\#$. This amounts to the following proof rule.

rule 5.4.0 (proof rule for finite lists): For predicate R we have $(0) \Leftarrow (1)$, with:

(0) $(\mathbf{A}\, x : L_\star\cdot x : R\cdot x)$

(1) $(\mathbf{A}\, x : L_\star\cdot x : (\mathbf{A}\, y : L_\star\cdot y \wedge \#y<\#x : R\cdot y) \Rightarrow R\cdot x)$.

□

Sometimes, properties of finite lists are proved by means of, so-called, *structural induction.* This is, however, nothing but a special case of the above rule.

theorem 5.4.1 (structural induction): For predicate R we have $(2) \Leftarrow (3)$, with:

(2) $(\mathbf{A}\, x : L_\star\cdot x : R\cdot x)$

(3) $R\cdot[\,] \wedge (\mathbf{A}\, a,x : L_\star\cdot x : R\cdot x \Rightarrow R\cdot(a\,;x))$.

proof: By derivation of (2) from (3) :

$(\mathbf{A}\, x : L_\star\cdot x : R\cdot x)$

$\Leftarrow$ { rule 5.4.0 }

$(\mathbf{A}\, x : L_\star\cdot x : (\mathbf{A}\, y : L_\star\cdot y \wedge \#y<\#x : R\cdot y) \Rightarrow R\cdot x)$

$=$ { corollary 5.2.8 for $L_\star$, with dummy renaming $b,y \leftarrow a,x$ }

$((\mathsf{A}\, y : L_*\cdot y \wedge \#y<\#[\,] : R\cdot y) \Rightarrow R\cdot[\,]) \wedge$
$(\mathsf{A}\, a,x : L_*\cdot x : (\mathsf{A}\, y : L_*\cdot y \wedge \#y<\#(a\,;x) : R\cdot y) \Rightarrow R\cdot(a\,;x)\,)$.

$=$ { $\#[\,]=0$, hence the first conjunct equals $R\cdot[\,]$, $\#(a\,;x)=\#x+1$ }

$R\cdot[\,] \wedge (\mathsf{A}\, a,x : L_*\cdot x : (\mathsf{A}\, y : L_*\cdot y \wedge \#y\leqslant\#x : R\cdot y) \Rightarrow R\cdot(a\,;x)\,)$

$\Leftarrow$ { instantiation: $y\leftarrow x$ }

$R\cdot[\,] \wedge (\mathsf{A}\, a,x : L_*\cdot x : R\cdot x \Rightarrow R\cdot(a\,;x)\,)$.

□

Each finite list is completely determined by its elements. The following theorem expresses this.

theorem 5.4.2: For finite lists x and y :

$$x=y \;\equiv\; \#x=\#y \wedge (\mathsf{A}\, i : 0\leqslant i<\#x : x\cdot i = y\cdot i) \quad .$$

proof: '$\Rightarrow$' : This is Leibniz,
'$\Leftarrow$' : By induction on $\#x$:

case $\#x=0$;

$\#x=\#y$

$=$ { $\#x=0$ }

$\#x=0 \wedge \#y=0$

$=$ { property 5.3.2 }

$x=[\,] \wedge y=[\,]$

$\Rightarrow$ { calculus }

$x=y$.

case $\#x>0$: on account of property 5.3.2 we may use $a\,;x$ and $b\,;y$ instead of x and y :

$a\,;x=b\,;y$

$=$ { property 5.1.4 }

$a=b \wedge x=y$

$\Leftarrow$ { property 5.3.2: $\#x<\#(a\,;x)$; induction hypothesis }

$a = b \wedge \#x = \#y \wedge (\mathbf{A}\, i : 0 \leq i < \#x : x{\cdot}i = y{\cdot}i)$

= { definition 5.1.1 }

$\#x = \#y \wedge (a\,;x){\cdot}0 = (b\,;y){\cdot}0 \wedge (\mathbf{A}\, i : 0 \leq i < \#x : (a\,;x){\cdot}(i+1) = (b\,;y){\cdot}(i+1))$

= { dummy substitution: $i+1 \leftarrow i$; range unsplit }

$\#x = \#y \wedge (\mathbf{A}\, i : 0 \leq i < \#x+1 : (a\,;x){\cdot}i = (b\,;y){\cdot}i)$

= { property 5.3.2: $\#x+1 = \#(a\,;x)$ }

$\#(a\,;x) = \#(b\,;y) \wedge (\mathbf{A}\, i : 0 \leq i < \#(a\,;x) : (a\,;x){\cdot}i = (b\,;y){\cdot}i)$.

□

5.5 Productivity theory and its application to infinite lists

5.5.0 introduction

Throughout this section we use *list* for *infinite list*, unless stated otherwise. In contrast to the, relatively simple, situation with finite lists, proving properties of infinite lists is more complicated. Because infinite lists cannot be ordered according to length, proofs by induction on their lengths are not possible. In our program notation, lists, and functions yielding them, can only be defined recursively. Therefore, in order to be able to prove properties of the values thus defined, we need some other induction principle. Of course, we can use definition 5.2.1 to base our proofs upon: the formula $(\mathbf{A}\, i : 0 \leq i : P_i{\cdot}x)$ suggests the use of mathematical induction over dummy i . It so happens, however, that definitions of lists, and of functions yielding lists, exhibit patterns for which a number of properties can be proved once and for all. For this purpose, we develop some theory.

The key notion in the theory developed here is *productivity*. As far as we have been able to trace, the first use of the term *productivity*, in connection with lists, occurs in [Dij1][Dij2]. The first attempt towards a formal definition and its use in a *productivity theorem* has been given in [Hoo0]. This has given rise to more general notions of productivity, such as the ones in [Boo] and [Sij].

The problem we are dealing with consists of two parts. First, we wish to know how to prove that the value of a given expression is a list. Second, given such an expression, the question is how to prove properties of that list. Notice that, because lists *are* functions on Nat , for the latter purpose all techniques for proving properties of functions are applicable. Conversely, the applicability of the theory developed here is not restricted to lists.

We formulate the definition of productivity and its associated theorems in a rather general way, without reference to lists. Next, by instantiation of this theory, we obtain a few special cases pertaining to lists and functions thereof. In chapter 6, we show how this theory can be used for the derivation of programs.

5.5.1 on equality of infinite lists

We consider the relation ω on L_∞ , defined by:

$(\omega\cdot)$ $\quad x \,\omega\, y \equiv (\mathbf{A}\, i : 0 \leqslant i : x\cdot i = y\cdot i)$.

This is an equivalence relation and lists x and y satisfying $x \,\omega\, y$ are, to all intents and purposes, functionally equivalent. Yet, in our formalism we cannot prove $x \,\omega\, y \Rightarrow x = y$. Nevertheless, we simply write $x = y$ instead of $x \,\omega\, y$, with the convention that, for infinite lists, $x = y$ means $x \,\omega\, y$.

In some cases, it is more convenient to interpret $x \,\omega\, y$ as follows:

$(\omega\uparrow)$ $\quad x \,\omega\, y \equiv (\mathbf{A}\, i : 0 \leqslant i : x\uparrow i = y\uparrow i)$.

$(\omega\cdot)$ and $(\omega\uparrow)$ are equivalent. Therefore, we use both without explicitly stating so at each occasion.

5.5.2 definitions

We start the development of the theory with the introduction of some notions. The theory is independent of the particular properties of our value domain Ω . The game played here takes place in an arbitrary set, the *universe*, that remains anonymous. Sets -- such as the U_i introduced below -- are subsets of this universe. The following properties form the starting point for the discussion; they are all we need for the development of the theory. Later, we apply the theory to sets U_i and relations ω_i that have much stronger properties than the ones formulated here. For example, the relations ω_i

happen to be equivalence relations. For the purpose of formulating the theory, however, we need not know this.

(0a) U_i is a set, for all i, $0 \leqslant i$
(0b) ω_i is a relation on U_i, for all i, $0 \leqslant i$
(1a) $(\mathbf{A}\, u :: U_0 \cdot u)$
(1b) $(\mathbf{A}\, u,v :: u\, \omega_0 v)$
(2a) $\ominus$ is a relation (on the universe)
(2b) $(\mathbf{A}\, u,v :: u \ominus v \Rightarrow (U_i \cdot u \equiv U_i \cdot v))$, for all i, $0 \leqslant i$
(2c) $(\mathbf{A}\, u,v,w :: u \ominus v \Rightarrow (u\, \omega_i w \equiv v\, \omega_i w) \wedge (w\, \omega_i u \equiv w\, \omega_i v))$, for all i, $0 \leqslant i$.

For the time being, the reader may read $=$ for $\ominus$. The only properties of $\ominus$ needed turn out to be (2b) and (2c) ; hence, our theorems may be applied in any situation satisfying (2b) and (2c). It so happens that we need this freedom in the applications of the theory.

definition 5.5.2.0 (definition of U_∞ and ω_∞):

$$(\mathbf{A}\, u :: U_\infty \cdot u \equiv (\mathbf{A}\, i :: U_i \cdot u))$$
$$(\mathbf{A}\, u,v : U_\infty \cdot u \wedge U_\infty \cdot v : u\, \omega_\infty v \equiv (\mathbf{A}\, i :: u\, \omega_i v))$$

□

definition 5.5.2.1 (productivity): "Function F is productive" $\equiv$ (3a) $\wedge$ (3b), with:

(3a) $(\mathbf{A}\, i,u :: U_i \cdot u \Rightarrow U_{i+1} \cdot (F \cdot u))$
(3b) $(\mathbf{A}\, u,v : U_\infty \cdot u \wedge U_\infty \cdot v : (\mathbf{A}\, i :: u\, \omega_i v \Rightarrow F \cdot u\, \omega_{i+1} F \cdot v))$

□

definition 5.5.2.2 (admissibility): "Predicate P is admissible" $\equiv$ "a sequence Q_i $(0 \leqslant i)$ of predicates exists satisfying (4a) $\wedge$ (4b)", with:

(4a) $(\mathbf{A}\, u : U_\infty \cdot u : P \cdot u \equiv (\mathbf{A}\, i :: Q_i \cdot u))$
(4b) $(\mathbf{A}\, u,v : U_\infty \cdot u \wedge U_\infty \cdot v : (\mathbf{A}\, i :: u\, \omega_i v \Rightarrow (Q_i \cdot u \equiv Q_i \cdot v)))$

□

property 5.5.2.3: For admissible predicates $P0$ and $P1$ predicate $P0 \wedge P1$ is admissible too.

□

convention 5.5.2.4 (productivity of definitions): Productive functions are used in recursive definitions. For productive F, we can define u by $u \ominus F \cdot u$. In practical situations, however, the definition of u is often formulated, without introduction of name F, by means of an expression E containing recursive occurrences of u; i.e, we simply write $u \ominus E$. This definition can be transformed into the former one by defining F by $F \cdot u \ominus E$. If function F, thus obtained, is productive we also call definition $u \ominus E$ productive.

□

5.5.3 theorems

We present two theorems. The first one states that all fixed points of a productive function are elements of U_∞, and that all such fixed points are "equivalent" in the sense of ω_∞. The second theorem provides sufficient conditions to be satisfied by a productive function and an admissible predicate, in order that all fixed points of the function satisfy the predicate. Neither theorem requires the function to have fixed points.

theorem 5.5.3.0 (first productivity theorem): Productive functions F satisfy $(5a) \wedge (5b)$, with:

(5a) $(\mathbf{A}\, u : u \ominus F \cdot u : U_\infty \cdot u)$

(5b) $(\mathbf{A}\, u,v : u \ominus F \cdot u \wedge v \ominus F \cdot v : u\ \omega_\infty\ v)$

proof: For u, satisfying $u \ominus F \cdot u$, we derive:

$U_\infty \cdot u$

$=$ { definition of U_∞ }

$(\mathbf{A}\, i :: U_i \cdot u)$

$\Leftarrow$ { mathematical induction }

$U_0 \cdot u \wedge (\mathbf{A}\, i :: U_i \cdot u \Rightarrow U_{i+1} \cdot u)$

$=$ { (1a), $u \ominus F \cdot u$ and (2b) }

$(\mathbf{A}\, i :: U_i \cdot u \Rightarrow U_{i+1} \cdot (F \cdot u))$

$=$ $\{$ F is productive: (3a) $\}$

true .

This proves (5a) . For u,v , satisfying $u \ominus F \cdot u \wedge v \ominus F \cdot v$, we now derive:

$u\, \omega_\infty v$

$=$ $\{$ definition of ω_∞ $\}$

$(\mathbf{A}\, i :: u\, \omega_i v)$

$\Leftarrow$ $\{$ mathematical induction $\}$

$u\, \omega_0 v \wedge (\mathbf{A}\, i :: u\, \omega_i v \Rightarrow u\, \omega_{i+1} v)$

$=$ $\{$ (1b) , $u \ominus F \cdot u \wedge v \ominus F \cdot v$ and (2c) $\}$

$(\mathbf{A}\, i :: u\, \omega_i v \Rightarrow F \cdot u\, \omega_{i+1} F \cdot v)$

$=$ $\{$ F is productive: (3b) $\}$

true .

Application of (3b) in the last step of this proof requires $U_\infty \cdot u \wedge U_\infty \cdot v$; we have: $U_\infty \cdot u \Leftarrow \{$ (5a) $\}$ $u \ominus F \cdot u$. This concludes the proof of (5b) .

□

theorem 5.5.3.1 (second productivity theorem): Productive functions F and admissible predicates P satisfy (6c) $\Leftarrow$ (6a) $\wedge$ (6b) , with:

(6a) $(\mathbf{E}\, u : U_\infty \cdot u : P \cdot u)$
(6b) $(\mathbf{A}\, u : U_\infty \cdot u : P \cdot u \Rightarrow P \cdot (F \cdot u))$
(6c) $(\mathbf{A}\, u : u \ominus F \cdot u : P \cdot u)$

proof: Assuming (6a) $\wedge$ (6b) we prove (6c) . For u , satisfying $u \ominus F \cdot u$, we construct a sequence $v_i (0 \leqslant i)$ -- of *approximations* of u -- , as follows:

(6d) v_0 such that $U_\infty \cdot v_0 \wedge P \cdot v_0$; on account of (6a) this is possible.
(6e) $v_{i+1} = F \cdot v_i$, $0 \leqslant i$

Notice that, on account of $u \ominus F \cdot u$, F's productivity, and (5a) , we have $U_\infty \cdot u$. Furthermore, we have (6f) $\wedge$ (6g) $\wedge$ (6h) , with:

(6f) $(\mathbf{A}\, i :: U_\infty\cdot v_i)$
(6g) $(\mathbf{A}\, i :: P\cdot v_i)$
(6h) $(\mathbf{A}\, i :: u\, \omega_i\, v_i)$

Both (6f) and (6g) are easily proved by mathematical induction. For (6f) , this requires, apart from the definitions of v and U_∞ , (3a) ; for (6g) , this requires (6b) . Moreover, (6h) is proved as follows:

$(\mathbf{A}\, i :: u\, \omega_i\, v_i)$
$\Leftarrow$ { mathematical induction }
$u\, \omega_0\, v_0 \wedge (\mathbf{A}\, i :: u\, \omega_i\, v_i \Rightarrow u\, \omega_{i+1}\, v_{i+1})$
$=$ { (1b) , $u = F\cdot u$ and (2c) , definition of v_{i+1} }
$(\mathbf{A}\, i :: u\, \omega_i\, v_i \Rightarrow F\cdot u\, \omega_{i+1}\, F\cdot v_i)$
$=$ { F is productive: (3b) , using $U_\infty\cdot u \wedge U_\infty\cdot v_i$ }
true .

□ (proof of (6h))

We now have what we need to prove $P\cdot u$:

$P\cdot u$
$=$ { $U_\infty\cdot u$, P is admissible: (4a) }
$(\mathbf{A}\, i :: Q_i\cdot u)$
$=$ { $U_\infty\cdot u$, (6f) , (6h) so: (4b) }
$(\mathbf{A}\, i :: Q_i\cdot v_i)$
$\Leftarrow$ { instantiation }
$(\mathbf{A}\, i :: (\mathbf{A}\, j :: Q_j\cdot v_i))$
$=$ { (6f) , P is admissible: (4a) }
$(\mathbf{A}\, i :: P\cdot v_i)$
$=$ { (6g) }
true .

This concludes the proof of this theorem.

□

5.5.4 list productivity

A simple application of the above theory is the following one. With P_i for U_i , $u\uparrow i = v\uparrow i$ for $u \sim_i v$, and $=$ for $\oplus$, we have $U_\infty = L_\infty$ and $\sim_\infty$ amounts to $=$ (in the sense of section 5.5.1). Then, conditions (0a) through (2c) are satisfied. Definition 5.5.2.1 amounts to the definition of, so-called, *list-productivity* .

definition 5.5.4.0: "Function F is list-productive" $\equiv$ (7a) $\wedge$ (7b) , with:

(7a) $(\mathbf{A}\, i,u :: P_i \cdot u \Rightarrow P_{i+1} \cdot (F \cdot u)\ )$
(7b) $(\mathbf{A}\, u,v : L_\infty \cdot u \wedge L_\infty \cdot v : (\mathbf{A}\, i :: u\uparrow i = v\uparrow i \ \Rightarrow\ F \cdot u \uparrow (i+1) = F \cdot v \uparrow (i+1)\ )\ )$

□

Similarly, definition 5.5.2.2 can be instantiated. Specifications of (infinite) lists often are predicates P satisfying:

(8) $(\mathbf{A}\, u : L_\infty \cdot u : P \cdot u \ \equiv\ (\mathbf{A}\, i :: Q_i \cdot (u\uparrow i))\ )$,

for some sequence Q_i $(0 \leqslant i)$ of predicates, where Q_i is a predicate on L_i . In words: infinite lists can often be specified in terms of their finite prefixes. According to the following lemma, such specifications are admissible.

lemma 5.5.4.1 (first admissibility lemma): (8) $\Rightarrow$ "P is admissible" .
proof: By straightforward application of definition 5.5.2.2, with $Q_i \circ (\uparrow i)$ for Q_i , and the above definitions of U_∞ and $\sim_\infty$.

□

example 5.5.4.2: For any value a , function $(a\,;)$ is list-productive; hence, by (5a) , $x|[x = a\,;x]|$ is an infinite list. Moreover, this x satisfies $P \cdot x$, with:

$P \cdot u \ \equiv\ (\mathbf{A}\, i :: u \cdot i = a)$.

In this simple case, $P \cdot x$ can be proved by straightforward mathematical induction. Generally, whenever the elements of the list have been specified explicitly in such a way that simple recurrence relations for these elements can be derived, such a "direct" proof by mathematical induction is simpler than a proof by an appeal to the second productivity theorem. The theorem is particularly useful in those cases where no such simple relations can be derived, for instance because the list's elements have been specified more implicitly. We present examples of this in chapter 6.

□

example 5.5.4.3: Predicate P is admissible, whereas $\neg P$ is not, with:

$$P{\cdot}u \equiv (\mathrm{A}\, i : 0 \leqslant i : u{\cdot}i = 1) \quad \text{, so}$$
$$\neg P{\cdot}u \equiv (\mathrm{E}\, i : 0 \leqslant i : u{\cdot}i \neq 1) \quad .$$

With F defined by $F{\cdot}x = 1 ; x$, both P and $\neg P$ satisfy conditions (6a) and (6b) of theorem 5.5.3.1; yet, $\neg P$ does not satisfy (6c) . This shows that the requirement of admissibility is not void.

□

5.5.5 uniform productivity

We consider the following definition of f , in terms of G and H :

(9a) $\quad f{\cdot}x \;=\; G{\cdot}x \; ; \; f{\cdot}(H{\cdot}x) \quad .$

For the sake of the discussion we rewrite (9a) into the equivalent:

(9b) $\quad f{\cdot}x \;=\; F{\cdot}f{\cdot}x$, where F is defined by

(9c) $\quad F{\cdot}g{\cdot}x \;=\; G{\cdot}x \; ; \; g{\cdot}(H{\cdot}x) \quad .$

Let X be a set. Throughout this subsection x ranges over X . We define:

$$\begin{aligned}
U_i &= (X \to \mathcal{P}_i) \\
u \sim_i v &\equiv (\mathrm{A}\, x :: u{\cdot}x{\uparrow}i = v{\cdot}x{\uparrow}i\,) \\
u \ominus v &\equiv (\mathrm{A}\, x :: u{\cdot}x = v{\cdot}x\,) \quad \text{, hence:} \\
U_\infty &= (X \to \mathcal{L}_\infty) \\
u \sim_\infty v &\equiv (\mathrm{A}\, x :: u{\cdot}x = v{\cdot}x\,) \quad .
\end{aligned}$$

Conditions (0a) through (2c) are met. Notice that, by the definition of $\ominus$

and (9b) , we now have $f \in F \cdot f$.

theorem 5.5.5.0 (uniform productivity theorem): With f defined by (9a) , we have (10b) $\Leftarrow$ (10a) , with:

(10a) H has type $X \to X$
(10b) f has type $X \to \mathcal{L}_\infty$

proof: By application of the first productivity theorem to function F . From this theorem we conclude $U_\infty \cdot f$, i.e: $(X \to \mathcal{L}_\infty) \cdot f$, provided that we prove that F is productive. Here, we prove (3a) only; the proof of (3b) has the same structure.

$$
\begin{array}{ll}
 & U_{i+1} \cdot (F \cdot u) \\
= & \quad \{ \text{ definition of } U_{i+1} \ \} \\
 & (X \to P_{i+1}) \cdot (F \cdot u) \\
= & \quad \{ \text{ definition of } \to \ \} \\
 & (\mathbf{A}\, x : X \cdot x : P_{i+1} \cdot (F \cdot u \cdot x)\ ) \\
= & \quad \{ \ (9c)\ \} \\
 & (\mathbf{A}\, x : X \cdot x : P_{i+1} \cdot (G \cdot x \, ; u \cdot (H \cdot x))\ ) \\
= & \quad \{ \ P_{i+1} \cdot (a \, ; x) \equiv P_i \cdot x \ \text{(property 5.2.6)} \ \} \\
 & (\mathbf{A}\, x : X \cdot x : P_i \cdot (u \cdot (H \cdot x))\ ) \\
\Leftarrow & \quad \{ \text{ predicate calculus, using (10a) } \} \\
 & (\mathbf{A}\, x : X \cdot x : P_i \cdot (u \cdot x)\ ) \\
= & \quad \{ \text{ definition of } \to \ \} \\
 & (X \to P_i) \cdot u \\
= & \quad \{ \text{ definition of } U_i \ \} \\
 & U_i \cdot u \quad .
\end{array}
$$

□

lemma 5.5.5.1 (second admissibility lemma): For every sequence Q_i $(0 \leq i)$ of predicates and for predicate P satisfying (11) , P is admissible, with:

(11) $(\mathbf{A}\, g : (X \to \mathcal{L}_\infty) \cdot g : P \cdot g \ \equiv \ (\mathbf{A}\, i, x :: Q_i \cdot x \cdot (g \cdot x \uparrow i)\ )\ )$

proof: Essentially the same as the proof of lemma 5.5.4.1.

□

Because the productivity of function F is independent of the properties of set X, we call F, and, hence, f's definition, *uniformly productive*. Notice that theorem 5.5.5.0 provides a, more or less, *synctactic* criterium for productivity; because, in our formalism, every function H has type $\Omega \to \Omega$, any definition of the form $f{\cdot}x = G{\cdot}x \; ; \; f{\cdot}(H{\cdot}x)$ yields a function f having type $\Omega \to \mathcal{L}_\infty$.

The theorem can be extended a little as follows. Under the additional requirement that G has type $X \to Y$, for some set Y, it is possible to prove that f has type $X \to \mathcal{L}_\infty(Y)$. This can be shown by a simple extension of the above proof of the theorem, or by proving $(\mathbf{A}\, x,i : X{\cdot}x \wedge 0 \leqslant i : Y{\cdot}(f{\cdot}x{\cdot}i))$ directly by mathematical induction on i.

example 5.5.5.2: We consider function from defined by:

$$from{\cdot}i \;=\; i \;;\; from{\cdot}(i+1) \quad .$$

This definition is of the form of (9a) ; therefore, it is uniformly productive. Observing that $i+1 = (+1){\cdot}i$ and that function $(+1)$ has type $Nat \to Nat$, we conclude, by application of theorem 5.5.5.0, that from has type $Nat \to \mathcal{L}_\infty$. Furthermore, it satisfies $(\mathbf{A}\, i,j : 0 \leqslant i \wedge 0 \leqslant j : from{\cdot}i{\cdot}j = i+j)$, which is easily proved by mathematical induction (on j).

□

5.5.6 non-uniform productivity

In this subsection, we consider the following definition of f, in terms of B, G, and H:

$$(12a) \quad f{\cdot}x \;=\; \begin{array}{l} (\;\; B{\cdot}x \to G{\cdot}x \;;\; f{\cdot}(H{\cdot}x) \\ \;\,[\!]\; \neg B{\cdot}x \to \qquad\quad f{\cdot}(H{\cdot}x) \\ ) \quad . \end{array}$$

Let X be a set. As before, dummy x ranges over X. We assume that B and H satisfy:

(12b) B has type $X \to Bool$
(12c) H has type $X \to X$.

We investigate under what conditions f has type $X \to L_\infty$. First, we observe that, for x satisfying $(\mathbf{A}\, i :: \neg B\cdot(H^i\cdot x))$, the only conclusion about f we can draw from (12a) is $f\cdot x = f\cdot(H^i\cdot x)$, for all i , which is not very useful. So, a necessary condition for the "usefulness" of definition (12a) is:

(13) $(\mathbf{A}\, x :: (\mathbf{E}\, i :: B\cdot(H^i\cdot x)))$.

We now show that (13) also is sufficient, by deriving an alternative definition for f that is uniformly productive, so that we can apply the uniform productivity theorem. Thus, we conclude that from (12a) through (13) it follows that f has type $X \to L_\infty$. For the remainder of this subsection, we assume that X satisfies (13) . We define function v , of type $X \to Nat$, as follows:

(14) $v\cdot x = (\mathbf{MIN}\, i : B\cdot(H^i\cdot x) : i)$.

That this is a correct definition follows from (13) and the fact that every non-empty subset of Nat has a smallest element. v has the following properties:

(15a) $(\mathbf{A}\, x :: B\cdot x \equiv v\cdot x = 0)$
(15b) $(\mathbf{A}\, x :: \neg B\cdot x \equiv v\cdot x = v\cdot(H\cdot x) + 1)$, hence
(15c) $(\mathbf{A}\, x :: v\cdot x > 0 \Rightarrow v\cdot(H\cdot x) < v\cdot x)$.

For the sake of the discussion, we introduce functions $F0$ and $F1$:

(16a) $F0\cdot g\cdot x = (\ B\cdot x \to G\cdot x \ ;\ g\cdot(H\cdot x)$
$\qquad\qquad\qquad [\!]\ \neg B\cdot x \to g\cdot(H\cdot x)$
$\qquad\qquad\qquad)$
(16b) $F1\cdot g\cdot x = G\cdot(H^k\cdot x) \ ;\ g\cdot(H^{k+1}\cdot x)\ [\![\ k = v\cdot x\ ]\!]$.

Definition (12a) can now be rewritten as $f\cdot x = F0\cdot f\cdot x$. Function $F1$ is of interest because it is uniformly productive. We now show, in a number of steps, that $(\mathbf{A}\, x :: f\cdot x = F1\cdot f\cdot x)$, so that we may apply the uniform productivity theorem (5.5.5.0) to f .

lemma 5.5.6.0: $(\mathbf{A}\, g,x :: F1{\cdot}g{\cdot}x \;=\; F0^k{\cdot}g{\cdot}x \;[\![\; k = v{\cdot}x+1 \;]\!]\;)$.

proof: By mathematical induction on the values of v :

case $v{\cdot}x = 0$:

$F0^k{\cdot}g{\cdot}x \;[\![\; k = v{\cdot}x+1 \;]\!]$

= { $v{\cdot}x = 0$ (unfolding k and where-clause elimination) }

$F0{\cdot}g{\cdot}x$

= { $v{\cdot}x = 0 \Rightarrow$ {(15a)} $B{\cdot}x$: (16a)(unfolding) }

$G{\cdot}x \;;\; g{\cdot}(H{\cdot}x)$

= { $v{\cdot}x = 0$ (where-clause introduction and folding k) }

$G{\cdot}(H^k{\cdot}x) \;;\; g{\cdot}(H^{k+1}{\cdot}x) \;[\![\; k = v{\cdot}x \;]\!]$

= { (16b) (folding) }

$F1{\cdot}g{\cdot}x$.

case $v{\cdot}x > 0$:

$F0^k{\cdot}g{\cdot}x \;[\![\; k = v{\cdot}x+1 \;]\!]$

= { dummy substitution: $k \leftarrow k+1$ }

$F0^{k+1}{\cdot}g{\cdot}x \;[\![\; k = v{\cdot}x \;]\!]$

= { $F0^{k+1}{\cdot}g = F0{\cdot}(F0^k{\cdot}g)$, $v{\cdot}x > 0 \Rightarrow$ {(15a)(15b)} $v{\cdot}x = v{\cdot}(H{\cdot}x)+1$ }

$F0{\cdot}(F0^k{\cdot}g){\cdot}x \;[\![\; k = v{\cdot}(H{\cdot}x)+1 \;]\!]$

= { $v{\cdot}x > 0 \Rightarrow$ {(15a)} $\neg B{\cdot}x$: (16a)(unfolding) with $g \leftarrow F0^k{\cdot}g$ }

$F0^k{\cdot}g{\cdot}(H{\cdot}x) \;[\![\; k = v{\cdot}(H{\cdot}x)+1 \;]\!]$

= { $v{\cdot}x > 0 \Rightarrow$ {(15c)} $v{\cdot}(H{\cdot}x) < v{\cdot}x$: induction hypothesis with $x \leftarrow H{\cdot}x$ }

$F1{\cdot}g{\cdot}(H{\cdot}x)$

= { (16b)(unfolding) }

$G{\cdot}(H^k{\cdot}(H{\cdot}x)) \;;\; g{\cdot}(H^{k+1}{\cdot}(H{\cdot}x)) \;[\![\; k = v{\cdot}(H{\cdot}x) \;]\!]$

= { $v{\cdot}x > 0 \Rightarrow v{\cdot}x = v{\cdot}(H{\cdot}x)+1$, so $k = v{\cdot}(H{\cdot}x) \equiv k+1 = v{\cdot}x$ }

$G{\cdot}(H^k{\cdot}(H{\cdot}x)) \;;\; g{\cdot}(H^{k+1}{\cdot}(H{\cdot}x)) \;[\![\; k+1 = v{\cdot}x \;]\!]$

= { $H^k{\cdot}(H{\cdot}x) = H^{k+1}{\cdot}x$, etcetera }

$G{\cdot}(H^{k+1}{\cdot}x) \;;\; g{\cdot}(H^{k+2}{\cdot}x) \;[\![\; k+1 = v{\cdot}x \;]\!]$

= { dummy substitution: $k+1 \leftarrow k$ }

$G\cdot(H^k\cdot x)\ ;\ g\cdot(H^{k+1}\cdot x)\ [\![\ k=v\cdot x\ ]\!]$

$=\qquad \{\ (16b)(\text{folding})\ \}$

$F1\cdot g\cdot x$.

□

lemma 5.5.6.1: $(\mathbf{A}\,k,x::\ f\cdot x=F0^k\cdot f\cdot x)$.

proof: By mathematical induction on k , using $f\cdot x=F0\cdot f\cdot x$ (obviously), and (16a) (unfortunately).

□

theorem 5.5.6.2: $(\mathbf{A}\,x::\ f\cdot x=F1\cdot f\cdot x)$.

proof:

$F1\cdot f\cdot x$

$=\qquad \{\ \text{lemma } 5.5.6.0\ \}$

$F0^k\cdot f\cdot x\ [\![\ k=v\cdot x+1\ ]\!]$

$=\qquad \{\ \text{lemma } 5.5.6.1\ \}$

$f\cdot x$.

□

corollary 5.5.6.3: f has type $X\to L_\infty$.

proof: Because f satisfies $(\mathbf{A}\,x::f\cdot x=F1\cdot f\cdot x)$, and because, as a consequence of (12c) , $H'\ [\![\ H'\cdot x=H^{k+1}\cdot x\ [\![\ k=v\cdot x\ ]\!]\ ]\!]$ has type $X\to X$, theorem 5.5.5.0 is applicable.

□

Similarly, we may use the second productivity theorem (5.5.3.1) to prove properties of f . Condition (6b) of this theorem then amounts to:

(17a) $(\mathbf{A}\,g:(X\to L_\infty)\cdot g:\ P\cdot g\Rightarrow P\cdot(F1\cdot g)\ )$.

This formula contains $F1$. It would be nice if we could reformulate (17a) in terms of $F0$, i.e. in terms of f's original definition, as follows:

(17b) $(\mathbf{A}\,g:(X\to L_\infty)\cdot g:\ P\cdot g\Rightarrow P\cdot(F0\cdot g)\ )$.

If P satisfies (17c), for some predicate R of type $X \to L_\infty \to \mathrm{Bool}$, this is indeed possible, as the following lemma shows.

(17c) $(\mathbf{A}g : (X \to L_\infty)\cdot g : P\cdot g \equiv (\mathbf{A}x :: R\cdot x\cdot(g\cdot x)))$.

lemma 5.5.6.4; For P satisfying (17c), we have (17a) $\Leftarrow$ (17b) .
proof:

$$
\begin{array}{ll}
 & P\cdot(F1\cdot g) \\
= & \quad \{ \text{ (17c) } \} \\
 & (\mathbf{A}x :: R\cdot x\cdot(F1\cdot g\cdot x)) \\
= & \quad \{ \text{ lemma 5.5.6.0, with } kx \text{ for } v\cdot x{+}1 \} \\
 & (\mathbf{A}x :: R\cdot x\cdot(F0^{kx}\cdot g\cdot x)) \\
\Leftarrow & \quad \{ \text{ instantiation } \} \\
 & (\mathbf{A}x :: (\mathbf{A}y :: R\cdot y\cdot(F0^{kx}\cdot g\cdot y))) \\
= & \quad \{ \text{ (17c) with } g \leftarrow F0^{kx}\cdot g \} \\
 & (\mathbf{A}x :: P\cdot(F0^{kx}\cdot g)) \\
\Leftarrow & \quad \{ \text{ instantiation } \} \\
 & (\mathbf{A}x :: (\mathbf{A}i :: P\cdot(F0^{i}\cdot g))) \\
\Leftarrow & \quad \{ \text{ elimination of the, now superfluous, dummy } x \} \\
 & (\mathbf{A}i :: P\cdot(F0^{i}\cdot g)) \quad .
\end{array}
$$

We have derived $(\mathbf{A}i :: P\cdot(F0^{i}\cdot g)) \Rightarrow P\cdot(F1\cdot g)$, which implies: $(\mathbf{A}g :: P\cdot g \Rightarrow (\mathbf{A}i :: P\cdot(F0^{i}\cdot g))) \Rightarrow (\mathbf{A}g :: P\cdot g \Rightarrow P\cdot(F1\cdot g))$. The antecedent of this implication follows from (17b) by mathematical induction.

□

Notice that predicates P satisfying condition (11) of the second admissibility lemma (5.5.5.1) are of the form (17c) . Hence, such predicates are also admissible for the application of theorem 5.5.3.1 to, so-called, *non-uniformly productive* definitions such as (12a) .

5.5.7 degrees of productivity

In this subsection we discuss list-productivity, cf. subsection 5.5.4, in greater detail. In order to be able to apply the productivity theorems we must prove that the function involved is productive. This can be done, of course, by means of the definition of productivity -- in our case: definition 5.5.4.0 -- , but this is rather awkward. Therefore, we develop a few rules by means of which the productivity of a function can be established more easily. The key to this is the observation that many functions can be considered as (function) compositions of standard functions. The productivity of these standard functions can be established once and for all, so all we need is a rule expressing the productivity of a composite function in terms of the productivities of its parts.

For this purpose, we generalise the (boolean) notion of list-productivity to one involving a function, of type $\mathrm{Int} \to \mathrm{Int}$, that represents the *degree of productivity* of the function it belongs to. This generalisation follows from the observation that $i+1$, in U_{i+1} and ω_{i+1} in the definition of productivity, equals $(+1)\cdot i$: the function $(+1)$ is just an instance of a whole class of functions. For our purpose, the *ascending* functions of type $\mathrm{Int} \to \mathrm{Int}$ suffice. In practical cases, these functions are usually of the kind $(+k)$, for integer k . Throughout this section we interpret $(-k)$ as the function $f [\![f\cdot x = x-k]\!]$, not as the number $-k$.

definition 5.5.7.0 (generalised list-productivity): For ascending function f , of type $\mathrm{Int} \to \mathrm{Int}$: "F is f-productive" $\equiv$ (18a) $\wedge$ (18b) , with:

(18a) $(\mathrm{A}\, i,x : 0 \leqslant i \wedge 0 \leqslant f\cdot i : P_i \cdot x \Rightarrow P_{f\cdot i} \cdot (F\cdot x))$

(18b) $(\mathrm{A}\, x,y : L_\infty \cdot x \wedge L_\infty \cdot y : (\mathrm{A}\, i : 0 \leqslant i \wedge 0 \leqslant f\cdot i : x\uparrow i = y\uparrow i \;\Rightarrow\; F\cdot x \uparrow f\cdot i = F\cdot y \uparrow f\cdot i))$

□

The relation between generalised list-productivity and list-productivity is given by the following property.

property 5.5.7.1: (21) $\Leftarrow$ (20) $\wedge$ (19) , with:

(19) $(\mathrm{A}\, i :: f\cdot i > i)$

(20) "F is f-productive"

(21) "F is list-productive"

□

property 5.5.7.2: For any a and natural j :

$I \,|[\, I \cdot x = x \,]|$ is $(+0)$-productive

$(a\,;\,)$ is $(+1)$-productive

$(\downarrow j)$ is $(-j)$-productive

$F \,|[\, F \cdot x = G \cdot (x \uparrow (j+1))\ ;\ F \cdot (x \downarrow 1) \,]|$ is $(-j)$-productive .

proof: We only show that $F \,|[\, F \cdot x = G \cdot (x \uparrow (j+1))\ ;\ F \cdot (x \downarrow 1) \,]|$ is $(-j)$-productive. For the proof of (18a) and (18b) the structure of the expression $G \cdot (x \uparrow (j+1))$ is largely irrelevant: we abbreviate it to Gx . With $(-j)$ for f , we have $f \cdot i = i - j$ and the formula $0 \leq i \wedge 0 \leq f \cdot i$ equivales $j \leq i$ (because $0 \leq j$). Both (18a) and (18b) can be proved by mathematical induction on i , with $j = i$ as the base case:

$P_0 \cdot (F \cdot x)$

$=$ { definition of P_0 }

true

$\Leftarrow$ { predicate calculus }

$P_j \cdot x$,

and, for $i : j \leq i$:

$P_{i+1-j} \cdot (F \cdot x)$

$=$ { unfolding F }

$P_{i+1-j} \cdot (Gx\,;\,F \cdot (x \downarrow 1))$

$=$ { $1 \leq i+1-j$; definition of P_{i+1-j} }

$P_{i-j} \cdot (F \cdot (x \downarrow 1))$

$\Leftarrow$ { induction hypothesis }

$P_i \cdot (x \downarrow 1)$

$=$ { $x = x \cdot 0\,;\,x \downarrow 1$; definition of P_{i+1} }

$P_{i+1} \cdot x$.

This proves (18a) ; similarly, we prove (18b) :

$F{\cdot}x{\uparrow}0 = F{\cdot}y{\uparrow}0$

$=$ {definition of $\uparrow 0$ }

true

$\Leftarrow$ { predicate calculus }

$x{\uparrow}j = y{\uparrow}j$,

and, for $i : j \leq i$:

$F{\cdot}x{\uparrow}(i+1-j) = F{\cdot}y{\uparrow}(i+1-j)$

$=$ { unfolding both applications of F , with Gy for $G{\cdot}(y{\uparrow}(j+1))$ }

$(Gx\,;F{\cdot}(x{\downarrow}1)){\uparrow}(i+1-j) = (Gy\,;F{\cdot}(y{\downarrow}1)){\uparrow}(i+1-j)$

$=$ { $1 \leq i+1-j$; definition of $\uparrow(i+1-j)$; property of ; (5.1.4) }

$Gx = Gy \wedge F{\cdot}(x{\downarrow}1){\uparrow}(i-j) = F{\cdot}(y{\downarrow}1){\uparrow}(i-j)$

$\Leftarrow$ { induction hypothesis }

$Gx = Gy \wedge (x{\downarrow}1){\uparrow}i = (y{\downarrow}1){\uparrow}i$

$=$ { definitions of Gx and Gy , $\uparrow\downarrow$-calculus (see below) }

$G{\cdot}(x{\uparrow}(j+1)) = G{\cdot}(y{\uparrow}(j+1)) \wedge x{\uparrow}(i+1){\downarrow}1 = y{\uparrow}(i+1){\downarrow}1$

$\Leftarrow$ { Leibniz (twice) }

$x{\uparrow}(j+1) = y{\uparrow}(j+1) \wedge x{\uparrow}(i+1) = y{\uparrow}(i+1)$

$=$ { $j \leq i$; (19b)(see below) }

$x{\uparrow}(i+1) = y{\uparrow}(i+1)$.

The hint "$\uparrow\downarrow$-calculus" refers to the property $x{\downarrow}j{\uparrow}i = x{\uparrow}(i+j){\downarrow}j$, for list x and natural i and j . This and other such properties are summarised in section 5.8.3.

□

theorem 5.5.7.3 (composition rule for productivity):

"F is f-productive" $\wedge$ "G is g-productive"
$\Rightarrow$ "$F{\circ}G$ is $(f{\circ}g)$-productive" .

proof: For f-productive F and g-productive G , we prove that the pair $F{\circ}G\,,f{\circ}g$ satisfies (18a) and (18b) . Both are proved by case analysis.

case $0 \leq i \wedge 0 \leq g{\cdot}i \wedge 0 \leq f{\cdot}(g{\cdot}i)$:

$\quad P_{f\cdot(g\cdot i)}\cdot(F\cdot(G\cdot x))$

$\Leftarrow \quad \{\ 0 \leq g{\cdot}i \wedge 0 \leq f{\cdot}(g{\cdot}i)$: F is f-productive $\}$

$\quad P_{g\cdot i}\cdot(G\cdot x)$

$\Leftarrow \quad \{\ 0 \leq i \wedge 0 \leq g{\cdot}i$: G is g-productive $\}$

$\quad P_i\cdot x$.

case $0 \leq i \wedge g{\cdot}i < 0 \wedge 0 \leq f{\cdot}(g{\cdot}i)$:

$\quad P_{f\cdot(g\cdot i)}\cdot(F\cdot(G\cdot x))$

$\Leftarrow \quad \{\ g{\cdot}i < 0$ and f is ascending, so: $0 \leq f{\cdot}(g{\cdot}i) \leq f{\cdot}0$: (19a)(see below) $\}$

$\quad P_{f\cdot 0}\cdot(F\cdot(G\cdot x))$

$\Leftarrow \quad \{\ 0 \leq 0 \wedge 0 \leq f{\cdot}0$: F is f-productive $\}$

$\quad P_0\cdot(G\cdot x)$

$= \quad \{$ (1a) $\}$

$\quad$ true

$\Leftarrow \quad \{$ predicate calculus $\}$

$\quad P_i\cdot x$.

This proves (18a) . The proof of (18b) has exactly the same structure. In it, (19b) (see below) is used.

□

The two properties used in the proof of this theorem are:

(19a) $(\mathsf{A}x :: P_j\cdot x \Leftarrow P_i\cdot x)$, for $i,j : 0 \leq j \leq i$

(19b) $(\mathsf{A}x,y :: x{\uparrow}j = y{\uparrow}j \Leftarrow x{\uparrow}i = y{\uparrow}i)$, for $i,j : 0 \leq j \leq i$.

Property (19a) follows directly from property 5.2.4, whereas (19b) follows from the following -- see section 5.8.3 -- property of $\uparrow$: $x{\uparrow}i{\uparrow}j = x{\uparrow}j$, for list x and natural $i,j : 0 \leq j \leq i$.

remark 5.5.7.4: Function composition of ascending functions is ascending.

□

Moreover, productivity is *monotonic*, in the following way.

theorem 5.5.7.5 (monotonicity of productivity): For f and g satisfying $(\mathbf{A}\, i : 0 \leq i : f{\cdot}i \geq g{\cdot}i)$, we have:

"F is f-productive" $\Rightarrow$ "F is g-productive" .

proof: By application of definition 5.5.7.0 and (19a) and (19b) .
□

example 5.5.7.6: We consider the following definitions:

$$\begin{aligned} F{\cdot}x &= (x{\cdot}0+x{\cdot}1) \;;\; F{\cdot}(x{\downarrow}1) \\ G{\cdot}x &= 0 \;;\; 1 \;;\; F{\cdot}x \\ s &= 0 \;;\; 1 \;;\; F{\cdot}s \quad . \end{aligned}$$

Function F is, on account of property 5.5.7.2, (-1)-productive; notice that, because $x{\uparrow}2 = [x{\cdot}0\,,x{\cdot}1]$, $x{\cdot}0+x{\cdot}1$ may be considered as a function of $x{\uparrow}2$. Function G can be considered as $(0\,;)\circ(1\,;)\circ F$; because $1+1-1=1$, G is, by the composition rule, $(+1)$-productive. As a result, we may apply the first productivity theorem and conclude $L_\infty{\cdot}s$. This being so, it is now possible to prove by mathematical induction over Nat that $(\mathbf{A}\, i\,;;\, s{\cdot}i = fib{\cdot}i)$, where fib has been defined in example 2.7.2.
□

example 5.5.7.7: We consider functions dup and half , defined by:

$$\begin{aligned} dup{\cdot}x &= x{\cdot}0 \;;\; x{\cdot}0 \;;\; dup{\cdot}(x{\downarrow}1) \\ half{\cdot}x &= x{\cdot}0 \;;\; half{\cdot}(x{\downarrow}2) \quad . \end{aligned}$$

dup is f-productive, whereas half is g-productive, with:

$$\begin{aligned} f{\cdot}i &= 2*i \\ g{\cdot}i &= (i+1)\ \mathbf{div}\ 2 \quad . \end{aligned}$$

Because $(f\circ g){\cdot}i \geq i$ and $(g\circ f){\cdot}i = i$, both dup∘half and half∘dup are $(+0)$-productive.
□

5.6 More operators on lists

In this section, we extend the program notation with two commonly used operators, viz. ++ ("cat") and rev ("reverse"). ++ is a binary operator; it is written in infix notation. rev is a (standard) function; in its applications we use normal function application. Both are redundant in the sense that they can be programmed in terms of the list primitives.

definition 5.6.0 (catenation): The binary operator ++ has the following types:

$$\begin{array}{lll} L_i \times L_j & \rightarrow L_{i+j} & \text{, for natural } i,j \\ L_\infty \times L & \rightarrow L_\infty & \\ L \times L_\infty & \rightarrow L_\infty & \text{, hence} \\ L \times L & \rightarrow L & . \end{array}$$

For lists x and y and finite list s , its value is defined by:

$$(\mathbf{A}\, i : 0 \leqslant i < \#x : (x{+\!\!+}y)\cdot i = x\cdot i)$$
$$(\mathbf{A}\, i : 0 \leqslant i < \#y : (s{+\!\!+}y)\cdot(\#s+i) = y\cdot i) \quad .$$

Notice that, from the above type indications, for finite lists s and t it follows that $\#(s{+\!\!+}t) = \#s + \#t$. Moreover, $\#(x {+\!\!+} y) = \infty \;\equiv\; \#x = \infty \vee \#y = \infty$. Syntactically, ++ has the same binding power as ; .

□

property 5.6.1: ++ is associative.
proof: By application of the above definition, with the use of theorem 5.4.2 and, for infinite lists, the notion of equality introduced in subsection 5.5.1.

□

property 5.6.2: [] is the identity element of ++

□

definition 5.6.3 (reverse): Function rev has type $L_i \rightarrow L_i$, for i , $0 \leqslant i$. For finite list s , its value is defined by:

$$(\mathbf{A}\, i : 0 \leqslant i < \#s : \text{rev}\cdot s\cdot i = s\cdot(\#s-1-i)) \quad .$$

□

5.7 The time complexity of the list operators

In this section we present the time complexities of the list operations. We give no other justification than that our choice is realistic to the extent that the program notation can be implemented in such a way that the requirements formulated here are met. It must be understood that the time complexities presented here pertain to the manipulation of the *structure* of lists only: the time needed for the evaluation of the *elements* of lists must be accounted for separately. The discussion of what time complexity means in the case of infinite lists is postponed until section 5.10.

postulate 5.7.0 (time complexity of the list operators): The following summary is a list of pairs E:T of expressions; each such pair must be read as: "evaluation of expression E takes O(T) time". In this summary, x denotes a list, s denotes a finite list, and i denotes a natural:

$[]$	:	1
$a;x$	:	1
$x=[]$	:	1
$x\cdot i$	:	i
$x\uparrow i$	:	i
$x\downarrow i$	:	i
$\#s$	:	$\#s$
$s+\!\!+x$	:	$\#s$
$rev\cdot s$	:	$\#s$

□

5.8 Algebraic properties of the list operators

The list operators have many algebraic properties that are useful for programming. In this section we summarise the ones that seem to be the more frequently used ones. Because, in applications, we need not distinguish between definitions and "derived" properties, we also repeat the definitions given earlier. Thus, we obtain a rather complete overview that can be used for reference purposes.

Throughout this section, a and b denote values -- of the element type, if so desired -- , x , y , z denote, either finite or infinite, lists, s and t denote finite lists, and i , j , k denote naturals. All formulae must be understood to be universally quantified over their free variables. All formulae, but one, are equalities that may, of course, be used in either "direction".

5.8.0 cons properties

$$
\begin{array}{lll}
a\,;x & \neq & [\,] \\
(a\,;x)\cdot 0 & = & a \\
(a\,;x)\cdot(i+1) & = & x\cdot i \\
a\,;[\,] & = & [a] \\
a\,;x = b\,;y & \equiv & a = b \wedge x = y
\end{array}
$$

5.8.1 cat properties

$$
\begin{array}{lll}
(x \mathbin{+\!\!+} y)\cdot i & = & x\cdot i \quad , \quad i < \# x \\
(s \mathbin{+\!\!+} y)\cdot(\# s+i) & = & y\cdot i \quad , \quad i < \# y \\
[\,] \mathbin{+\!\!+} x & = & x \\
x \mathbin{+\!\!+} [\,] & = & x \\
(a\,;x) \mathbin{+\!\!+} y & = & a\,;(x \mathbin{+\!\!+} y) \\
[a] \mathbin{+\!\!+} x & = & a\,;x \\
x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) & = & (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \\
x \mathbin{+\!\!+} y = [\,] & \equiv & x = [\,] \wedge y = [\,] \\
x \mathbin{+\!\!+} y = x & \equiv & L_{\infty}\cdot x \vee y = [\,]
\end{array}
$$

5.8.2 rev properties

$$
\begin{array}{lll}
rev\cdot s\cdot i & = & s\cdot(\# s-1-i) \quad , \quad i < \# s \\
rev\cdot[\,] & = & [\,] \\
rev\cdot[a] & = & [a] \\
rev\cdot(a\,;s) & = & rev\cdot s \mathbin{+\!\!+} [a] \\
rev\cdot(s \mathbin{+\!\!+} [a]) & = & a\,;rev\cdot s \\
rev\cdot(s \mathbin{+\!\!+} t) & = & rev\cdot t \mathbin{+\!\!+} rev\cdot s \\
rev\cdot(rev\cdot s) & = & s
\end{array}
$$

5.8.3 take and drop properties

$$\left.\begin{array}{lll} (x\uparrow i)\cdot j & = x\cdot j & ,\ j < k \\ (x\downarrow i)\cdot j & = x\cdot(k+j) & ,\ k+j < \#x \end{array}\right\}\ \text{where}\ \ k = i\ \mathbf{min}\ \#x$$

$$\begin{array}{ll} x\uparrow 0 & = [\,] \\ x\downarrow 0 & = x \\ [\,]\uparrow i & = [\,] \\ [\,]\downarrow i & = [\,] \\ (a\,;x)\uparrow(i+1) & = a\ ;\ x\uparrow i \\ (a\,;x)\downarrow(i+1) & = x\downarrow i \end{array}$$

$$\begin{array}{lll} (x +\!\!+ y)\uparrow i & = x\uparrow i & ,\ i \leqslant \#x \\ (x +\!\!+ y)\uparrow i & = x +\!\!+ y\uparrow(i-\#x) & ,\ \#x < i \\ (x +\!\!+ y)\downarrow i & = x\downarrow i +\!\!+ y & ,\ i \leqslant \#x \\ (x +\!\!+ y)\downarrow i & = y\downarrow(i-\#x) & ,\ \#x < i \end{array}$$

$$\begin{array}{lll} x & = x\uparrow i\ +\!\!+\ x\downarrow i & \\ x\uparrow(i+j) & = x\uparrow i\ +\!\!+\ x\downarrow i\uparrow j & \\ x\downarrow(i+j) & = x\downarrow i\downarrow j & \\ x\uparrow(i+j)\uparrow i & = x\uparrow i & \\ x\uparrow(i+j)\downarrow i & = x\downarrow i\uparrow j & \\ rev\cdot(x\uparrow(i+j))\uparrow j & = rev\cdot(x\downarrow i\uparrow j) & ,\ i+j \leqslant \#x \\ rev\cdot(x\uparrow(i+j))\downarrow j & = rev\cdot(x\uparrow i) & ,\ i+j \leqslant \#x \end{array}$$

5.8.4 # properties

$$\begin{array}{ll} \#[\,] & = 0 \\ \#[a] & = 1 \\ \#(a\,;s) & = 1 + \#s \\ \#(s +\!\!+ t) & = \#s + \#t \\ \#(rev\cdot s) & = \#s \\ \#(x\uparrow i) & = \#x\ \mathbf{min}\ i \\ \#(s\downarrow i) & = (\#s - i)\ \mathbf{max}\ 0 \end{array}$$

5.9 Definition and parameter patterns for lists

In section 2.7 we have introduced definition and parameter patterns for naturals and for tuples. Similarly, we now introduce such patterns for lists. Because tuples are finite lists, we must see to it that the definitions given here are consistent with the corresponding definition for tuples. As in section 2.7, we give informal definitions only, by means of examples.

definition 5.9.0 (definition patterns for lists): We give a definition for lists of length 3 and for listoids of order 3 only. For expression E , names a , b , c , s :

$$\begin{array}{rll} [a,b,c] = E & \text{means} & a = E{\cdot}0 \ \& \ b = E{\cdot}1 \ \& \ c = E{\cdot}2 \\ a\,;b\,;c\,;s = E & \text{means} & a = E{\cdot}0 \ \& \ b = E{\cdot}1 \ \& \ c = E{\cdot}2 \ \& \ s = E{\downarrow}3 \end{array} \quad .$$

Notice that the definition $[a,b,c] = E$ implies the equality $[a,b,c] = E$ *only* if $L_3{\cdot}E$. Similarly, the definition $a\,;b\,;c\,;s = E$ implies the corresponding equality *only* if $P_3{\cdot}E$. The definition patterns introduced here are intended to be used for these cases only.

□

definition 5.9.1 (parameter patterns for lists): We give a definition for lists of length 0 and 3 , and for listoids of order 3 only. For expression E , names a , b , c , s , and fresh name x :

$$\begin{array}{rll} f{\cdot}[\,] = E & \text{means} & f{\cdot}x \ = \ (\#x = 0 \ \rightarrow \ E) \\ f{\cdot}[a,b,c] = E & \text{means} & f{\cdot}x \ = \ (\#x = 3 \ \rightarrow \ E[\![\, [a,b,c] = x \,]\!]) \\ f{\cdot}(a\,;b\,;c\,;s) = E & \text{means} & f{\cdot}x \ = \ (\#x \geqslant 3 \ \rightarrow \ E[\![\, a\,;b\,;c\,;s = x \,]\!]) \end{array} \quad .$$

In these definitions we have used the # operator, which is defined for finite lists only. We wish, however, to use parameter patterns in definitions of functions on L_∞ too, but #x is not defined for listoids that are not finite lists. Yet, it is possible to construct boolean expressions for $\#x = i$ and $\#x \geqslant i{+}1$, for natural i and x satisfying $L_*{\cdot}x \vee P_{i+1}{\cdot}x$, in such a way that $P_{i+1}{\cdot}x \Rightarrow \neg(\#x = i)$ and $P_{i+1}{\cdot}x \Rightarrow \#x \geqslant i{+}1$. First, we observe that $\#x = i \ \equiv \ \#x \geqslant i \wedge \neg(\#x \geqslant i{+}1)$. Second, for $\#x \geqslant 0$ we may use true , and for $\#x \geqslant i{+}1$ we may use $x{\downarrow}i \neq [\,]$. Notice that these expressions have different time complexities: evaluation of $\#x \geqslant i{+}1$ takes $O(\#x)$ time, whereas evaluation of $x{\downarrow}i \neq []$ takes $O(\#x \,\mathbf{min}\, i)$ time.

□

example 5.9.2: Without the use of parameter patterns, a function sum, mapping finite lists over Int to the sums of their elements, can be defined as follows:

```
sum·s   =  ( s = [] → 0
           ▯ s ≠ [] → s·0 + sum·(s↓1)
           )   .
```

Using parameter patterns, we can rewrite this definition into:

```
   sum·[]      =  0
 & sum·(a ; s) =  a + sum·s    .
```

□

example 5.9.3: A function sqr that squares the elements of a, finite or infinite, list over Int can be defined as follows:

```
   sqr·[]      =  []
 & sqr·(a ; s) =  (a ⋆ a) ; sqr·s
```

□

The use of parameter patterns in the definition of a function influences the productivity of that function. This is caused by the presence of the guards implied by the parameter pattern. We illustrate this by means of an example.

example 5.9.4: We consider functions F and G defined by:

```
F·x            =  x       ,
G·(a ; b ; s)  =  a ; b ; s    .
```

These definitions are not equivalent: F is (+0)-productive, wheras G is (−1)-productive but not (+0)-productive. We show this by analysing G's definition. According to definition 5.9.1, this definition is equivalent to:

```
G·x  =  ( x↓1 ≠ [] → a ; b ; s ⟦ a ; b ; s = x ⟧ )    .
```

On account of the proof rule for guarded selections, this definition is only meaningful for x satisfying $x{\downarrow}1 \neq [\,]$; i.e, only for such x we can prove properties of $G{\cdot}x$. We now derive:

$$\begin{array}{ll} & x{\downarrow}1 \neq [\,] \\ \Leftarrow & \quad \{\ (\mathbf{A}\, y : P_1{\cdot}y : y \neq [\,])\ \} \\ & P_1{\cdot}(x{\downarrow}1) \\ \Leftarrow & \quad \{\ (\mathbf{A}\, y : P_2{\cdot}y : P_1{\cdot}(y{\downarrow}1))\ \} \\ & P_2{\cdot}x \quad . \end{array}$$

Both implications in this derivation are unavoidable; for instance, P_1 is the *largest* subset of Ω whose elements are, with our proof rules, provably different from $[\,]$. Hence, the best we can prove is:

$$(\mathbf{A}\, x : P_2{\cdot}x : x{\downarrow}1 \neq [\,]) \quad .$$

Consequently, we cannot prove that G has type $P_1 \rightarrow P_1$; so, we cannot conclude that G is $(+0)$-productive. The best we can try to prove is that it has type $P_2 \rightarrow P_1$, which indeed is possible.

The effect of this phenomenon increases with the length of the parameter pattern. In practice, this is hardly annoying, but it means that the use of parameter patterns requires some care, especially when used in definitions of functions that are intended to be productive.

□

5.10 On the time complexity of infinite-list programs

Expressions whose values are infinite lists cannot be evaluated in a finite amount of time. Yet, they are useful because they can, by means of the $\uparrow$ operator, be mapped onto finite lists. This enables us to separate specification and derivation of an infinite-list expression from the way it will be used, thus enhancing modularisation. We discuss an example of this in chapter 9.

We define the time complexity of infinite-list expressions as the time complexity of their finite prefixes, as follows.

definition 5.10.0 (time complexity of infinite lists): For infinite-list expression E , its time complexity is the time complexity of $E\uparrow i$, as a function of i . For example, we call E's time complexity *linear* or *quadratic*, if evaluation of $E\uparrow i$ requires $O(i)$ or $O(i^2)$ time respectively.

□

postulate 5.10.1: Let F be list-productive and let Tf be F's time complexity, such that evaluation of $F{\cdot}x\uparrow i$ takes $Tf{\cdot}i$ time, for x , $L_\infty{\cdot}x$, and i , $0 \leqslant i$. Then, evaluation of $X\uparrow i$ also takes $Tf{\cdot}i$ time, with:

$$X = F{\cdot}X \quad .$$

□

In practice, this postulate means that when we determine the time complexity of a recursively defined infinite list, we need not take into account the recursive occurrences of that list.

example 5.10.2: From example 5.5.7.6 we recall the definition of the infinite list of Fibonacci numbers:

$$\begin{aligned} &F{\cdot}x = (x{\cdot}0+x{\cdot}1) \;;\; F{\cdot}(x\downarrow 1) \\ \& \; &s \;\;\; = 0 \;;\; 1 \;;\; F{\cdot}s \quad . \end{aligned}$$

With $Tf{\cdot}i$ for the time needed to evaluate $F{\cdot}x\uparrow i$, we obtain from this definition the following recurrence relations for Tf ; here, we have counted unfoldings of F only:

$$\begin{aligned} Tf{\cdot}0 &= 0 \quad , \\ Tf{\cdot}(i+1) &= 1 + Tf{\cdot}i \quad . \end{aligned}$$

So, we have $Tf{\cdot}i = O(i)$; by application of postulate 5.10.1, we conclude that s has linear time complexity too.

□

5.11 Discussion

We discuss a few aspects of the theory developed in this chapter in relation to some other recent work on this subject.

Our definition of lists enables us to interpret lists, depending on our needs, in several different ways. This freedom is particularly useful for the construction of specifications and for proving properties of the objects thus specified. The usual recursive definition of lists is well-suited for use in programs, because this definition admits a simple operational interpretation. In specifications, however, the use of recursion often leads to overspecification, i.e. the specification contains more information than desirable. In chapter 11, we discuss the proper role of specifications more extensively. Here, we illustrate, by means of a few examples, the use of the other two interpretations of lists. All examples discussed here pertain to finite lists.

Lists are *functions*. Thus, a function mapping a list over Int to the sum of its elements can be specified by:

$$\text{sum}\cdot x \;=\; (\mathbf{S}\, i : 0 \leqslant i < \#x : x\cdot i) \qquad .$$

Similarly, function rev can be specified by:

$$\#(\text{rev}\cdot x) = \#x \;\wedge\; (\mathbf{A}\, i : 0 \leqslant i < \#x : \text{rev}\cdot x\cdot i = x\cdot(\#x-1-i)\,) \qquad .$$

With this specification, it is a trivial exercise to prove that, for finite list x, $\text{rev}\cdot(\text{rev}\cdot x) = x$; when the recursive definition of rev is used this proof requires the invention of an auxiliary property [Bir0]. Many specifications can be formulated easily in terms of the functional interpretation of lists. A disadvantage is that the manipulation of these specifications may give rise to a large amount of -- laborious and error prone -- *subscript juggling.*

Lists can be defined *algebraically*, in terms of the associative operator $+\!\!+$ and its identity element $[]$. For example, rev can be specified algebraically as follows:

$$\begin{aligned}
\text{rev}\cdot[] &= [] \quad ,\\
\text{rev}\cdot[a] &= [a] \quad ,\\
\text{rev}\cdot(x +\!\!+ y) &= \text{rev}\cdot y +\!\!+ \text{rev}\cdot x \quad .
\end{aligned}$$

As a second example, a, so-called, *segment* of list x can be defined to be a list t such that $x = s \mathbin{+\!\!+} t \mathbin{+\!\!+} u$, for some lists s , u . Thus, the minimal sum over all segments of list x can be specified by:

$$(\mathbf{MIN}\, s,t,u : x = s \mathbin{+\!\!+} t \mathbin{+\!\!+} u : sum \cdot t) \quad ,$$

where sum is the function specified above. The advantage of the algebraic approach is that subscript juggling can be largely avoided; its disadvantage is that it is sometimes less obvious -- see the example of rev -- that the specification captures our intentions. The, so-called, *Bird/Meertens style* of function programming [Bir1] is an example of a consistent elaboration of the algebraic approach. The above example shows that functional and algebraic formulations can be used in one and the same specification. We believe that this is more efficient than strict adherence to either of the two styles.

There is a remarkable correspondence between our theory of infinite lists and the elementary theory of *processes* developed in [Hoa1]. In [Hoa1], processes are defined by means of infinite sequences that are similar to infinite lists. The notions of *constructivity* and *nondestructivity*, as defined in [Hoa1], correspond to (+1)-productivity and (+0)-productivity respectively. The two composition rules for nondestructive functions and for constructive functions are instances of theorem 5.5.7.3.

The productivity theorems developed in this chapter allow several generalisations. A simple one is that the condition "F is productive" may be weakened to $(\mathbf{E}\, i : 1 \leqslant i : \text{"}F^i \text{ is productive"})$. This generalisation enables us, for instance, to discuss function F defined by:

$$F \cdot x \;=\; 0 \;;\; x \cdot 2 \;;\; x \quad .$$

F is not productive, but F^2 is productive and, indeed, $x = F \cdot x$ implies $L_\infty \cdot x$ and $(\mathbf{A}\, i :: x \cdot i = 0)$. In [Sij], more general notions of productivity, such as *set productivity*, and more general productivity theorems are formulated. For applications to program development, we think that the simpler versions are sufficient and more easy to use.

6 Programming with lists

6.0 Introduction

In this chapter we show, by means of a number of small examples, how the theory developed in chapter 5 can be applied. We do so in a formal way, but this is rather laborious. In later chapters, therefore, we use this theory more implicitly. The purpose of this chapter is to bridge the gap between theory and practice. Furthermore, in the last two sections of this chapter, we discuss the influence of the use of lists on the time complexity of programs.

6.1 Programs for rev

In this section, we illustrate the development of programs involving finite lists by the derivation of programs for the (standard) function rev . As starting point we use the following specification, which, for the sake of manipulability, has been split into 3 parts. Dummy s ranges over $L_\star$:

(0a) $L_\star\cdot(\text{rev}\cdot s)$

(0b) $\#(\text{rev}\cdot s) = \#s$

(0c) $(\mathbf{A}\, i : 0 \leq i < \#s : \text{rev}\cdot s\cdot i = s\cdot(\#s-1-i))$.

We derive a program by induction on $\#s$. On account of property 5.3.2, the case $\#s=0$ corresponds to $s=[]$, and the case $\#s>0$ corresponds to $s=b;t$, for some b and list t with $\#t<\#s$. Therefore, we usually distinguish cases [] and b;t right away: often, induction on $\#s$ boils down to structural induction only.

For the case $s=[]$, part (0c) of the specification is satisfied independently of what we choose for rev·[] . From (0a) and (0b) we infer that rev·[] must be a list of length 0 ; because the only list of length 0 is [] , we have no choice here. For the case $s=b;t$ we derive, for $i : 0 \leq i < \#(b;t)$:

$rev\cdot(b\,;t)\cdot i$

$=$ { specification of rev, part (0c) }

$(b\,;t)\cdot(\#(b\,;t)-1-i)$

$=$ { $\#(b\,;t)=\#t+1$ (property 5.3.2) }

$(b\,;t)\cdot(\#t-i)$.

This formula is amenable to application of the definition of ; ; this requires case analysis.

case $i=\#t$:

$(b\,;t)\cdot(\#t-\#t)$

$=$ { $\#t-\#t=0$, (5.1.1) }

b .

case $i<\#t$:

$(b\,;t)\cdot(\#t-i)$

$=$ { $\#t-i>0$, definition of ; }

$t\cdot(\#t-1-i)$

$=$ { induction hypothesis: (0c) for t, using $0\leqslant i<\#t$ }

$rev\cdot t\cdot i$.

Thus, we conclude that the specification for $rev\cdot(b\,;t)$ is satisfied by a value u , provided that:

(1a) $L_{*}\cdot u$

(1b) $\#u=\#t+1$

(1c) $u\cdot(\#t)=b$

(1d) $(\mathrm{A}\,i : 0\leqslant i<\#t : u\cdot i=rev\cdot t\cdot i\ )$.

Part (1d) can be satisfied by choosing $u=rev\cdot t+\!\!+v$, for some v ; from (1a) and (1b) it then follows that v must be a list of length 1 . Moreover, in order to satisfy (1c) , we need:

$$
\begin{aligned}
& (\mathrm{rev}{\cdot}t + v){\cdot}(\#t) = b \\
= \quad & \{\ \#(\mathrm{rev}{\cdot}t) = \#t\ (\text{ by induction hypothesis})\ \} \\
& (\mathrm{rev}{\cdot}t + v){\cdot}(\#(\mathrm{rev}{\cdot}t)) = b \\
= \quad & \{\ \text{definition of } + \ (5.6.0)\ \} \\
& v{\cdot}0 = b \quad .
\end{aligned}
$$

So, for v we can only choose [b] . Using parameter patterns, the definition for rev can now be encoded as follows:

```
    rev·[]     = []
  & rev·(b;t)  = rev·t ++ [b]    .
```

The time complexity of the above definition is quadratic in the size of the list: because #(rev·t) = #t , evaluation of the expression rev·t ++ [b] takes O(#t) time for the ++ operation, plus the time needed for the evaluation of rev·t itself. By means of the Tail Recursion Theorem, this definition can be transformed into the following one -- notice that ++ is associative, that it has [] as identity element, and that [b]++x = b;x -- :

```
    rev = rev1·[] |[ rev1·x·[]     = x
                  & rev1·x·(b;t) = rev1·(b;x)·t
                  ]|   .
```

As a result of this transformation, the "expensive" use of ++ has been eliminated. As a result, the time complexity of rev1·x·s is O(#s) .

6.2 A tail recursion theorem for lists

The Tail Recursion Theorem in chapter 4 pertains to functions defined in terms of an associative operator ⊕ with left identity e . Here, we present a similar theorem for functions on finite lists that does not require associativity of the operator involved, at the expense of an occurrence of function rev . The theorem has been invented by J.N.E. Bos [Hoo1]; it also occurs in [Bir0] under the name *third duality theorem.*

The theorem states the relation between functions F and G , defined on L_* by:

$$
\begin{array}{lll}
(0) & \quad F\cdot[] & = X \\
& \& \ F\cdot(a;s) & = a \oplus F\cdot s \\
(1) & \quad G\cdot x\cdot[] & = x \\
& \& \ G\cdot x\cdot(a;s) & = G\cdot(a\oplus x)\cdot s \quad .
\end{array}
$$

theorem 6.2.0 (Tail Recursion Theorem for lists): Functions F and G defined by (0) and (1) satisfy:

(2) $\quad (\mathbf{A}\, s : L_*\cdot s : F\cdot(\mathrm{rev}\cdot s) = G\cdot X\cdot s) \quad .$

proof: Actually, we prove a more general theorem, namely:

(3) $\quad (\mathbf{A}\, s,t : L_*\cdot s \wedge L_*\cdot t : F\cdot(\mathrm{rev}\cdot s +\!\!\!+ t) = G\cdot(F\cdot t)\cdot s) \quad .$

(2) follows from (3) by the instantiation $t \leftarrow []$, because $F\cdot[] = X$ and $\mathrm{rev}\cdot s +\!\!\!+ [] = \mathrm{rev}\cdot s$. We now prove (3) by induction on s :

$$
\begin{array}{ll}
& F\cdot(\mathrm{rev}\cdot[] +\!\!\!+ t) \\
= & \quad \{\ \mathrm{rev}\cdot[] = [] \text{ and } [] +\!\!\!+ t = t\ \} \\
& F\cdot t \\
= & \quad \{\ (1)\ (\text{folding})\ \} \\
& G\cdot(F\cdot t)\cdot[] \quad .
\end{array}
$$

Furthermore:

$$
\begin{array}{ll}
& F\cdot(\mathrm{rev}\cdot(a;s) +\!\!\!+ t) \\
= & \quad \{\ \mathrm{rev}\cdot(a;s) = \mathrm{rev}\cdot s +\!\!\!+ [a]\ ,\ +\!\!\!+ \text{ is associative}\ \} \\
& F\cdot(\mathrm{rev}\cdot s +\!\!\!+ ([a] +\!\!\!+ t)) \\
= & \quad \{\ \text{induction hypothesis}\ ,\ [a] +\!\!\!+ t = a;t\ \} \\
& G\cdot(F\cdot(a;t))\cdot s \\
= & \quad \{\ (0)\ (\text{unfolding})\ \}
\end{array}
$$

$$\begin{array}{ll} & G\cdot(a\oplus F\cdot t)\cdot s \\ = & \quad \{\ (1)\ (\text{folding})\ \} \\ & G\cdot(F\cdot t)\cdot(a\,;s) \quad . \end{array}$$

□

For example, with [] for X and ; for ⊕ , function F , as defined by (0) , satisfies $(\mathbf{A}\,s : L_*\cdot s : F\cdot s = s)$, whereas definition (1) amounts to

$$\begin{array}{ll} & G\cdot x\cdot[\,] \quad\ = x \\ \& & G\cdot x\cdot(a\,;s) = G\cdot(a\,;x)\cdot s \quad . \end{array}$$

Application of theorem 6.2.0 yields $(\mathbf{A}\,s ; L_*\cdot s : rev\cdot s = G\cdot[\,]\cdot s)$. Actually, we now have derived the same definition for rev , with G being rev1 , as in the previous section.

It is possible to derive sufficient conditions, to be imposed upon ⊕ , such that $F\cdot(rev\cdot s) = F\cdot s$, for all s [Hoo1]. Thus, a special case of the above theorem is obtained that does not contain rev anymore. This version of the theorem turns out to be a special case of a theorem by D.C. Cooper [Coo].

6.3 The map and zap operators

For given finite or infinite list x and function f , we consider the list, with the same length as x , that is obtained by application of f to each element of x . A function map that takes f and x as parameters and that yields this list can be specified as follows, with x ranging over L :

(0a) $L\cdot(map\cdot f\cdot x)$
(0b) $\#(map\cdot f\cdot x) = \#x$
(0c) $(\mathbf{A}\,i : 0 \leqslant i < \#x : map\cdot f\cdot x\cdot i = f\cdot(x\cdot i))$.

We derive a recursive definition for map . For the time being, we only consider finite lists. Because of (0a) and (0b) , the case [] (for x) leaves us no other possibility than the choice $map\cdot f\cdot[\,] = [\,]$. This satisfies (0c) . For the case a ; x , we observe that, on account of (0a) and (0b) , $map\cdot f\cdot(a\,;x)$ must be a nonempty list. Therefore, a good strategy is to try to define $map{\cdot}f{\cdot}(a\,;x)$ as $b\,;y$, by deriving expressions for b and y separately. Notice that this amounts to $b = map{\cdot}f{\cdot}(a\,;x){\cdot}0$ and $(\mathbf{A}\, i : 0 \leq i < \#x : y{\cdot}i = map{\cdot}f{\cdot}(a\,;x){\cdot}(i+1))$. Therefore, we carry out the following derivation:

$$
\begin{array}{ll}
 & map{\cdot}f{\cdot}(a\,;x){\cdot}0 \\
= & \quad \{\ (0c)\ \} \\
 & f{\cdot}((a\,;x){\cdot}0) \\
= & \quad \{\ \text{definition of } ;\ \} \\
 & f{\cdot}a \\
= & \quad \{\ \text{definition of } ; \text{, using } ? \text{ to denote a value irrelevant here}\ \} \\
 & (f{\cdot}a\ ;\ ?){\cdot}0 \quad ,
\end{array}
$$

The last step of this derivation serves to obtain an expression of the form $(f{\cdot}a\,;y){\cdot}0$: we now have derived that $map{\cdot}f{\cdot}(a\,;x){\cdot}0$ should be equal to $(f{\cdot}a\,;y){\cdot}0$, for some y to be chosen later; this requirement can be satisfied by the definition $map{\cdot}f{\cdot}(a\,;x) = f{\cdot}a\,;y$. In order to avoid the introduction of a name for a value that, for this part of the derivation, is irrelevant, we have used $?$ to denote that value. Next, we derive, for $i : 0 \leq i < \#x$:

$$
\begin{array}{ll}
 & map{\cdot}f{\cdot}(a\,;x){\cdot}(i+1) \\
= & \quad \{\ (0c)\ \} \\
 & f{\cdot}((a\,;x){\cdot}(i+1)) \\
= & \quad \{\ \text{definition of } ;\ \} \\
 & f{\cdot}(x{\cdot}i) \\
= & \quad \{\ \text{induction hypothesis: } (0c)\ \} \\
 & map{\cdot}f{\cdot}x{\cdot}i \\
= & \quad \{\ \text{definition of } ; \text{, using } ? \text{ to denote (another) irrelevant value}\ \} \\
 & (?\ ;\ map{\cdot}f{\cdot}x){\cdot}(i+1) \quad .
\end{array}
$$

The purpose of each last step in these two derivations is to obtain formulae that, like the formulae we started with, end with ${\cdot}0$ and ${\cdot}(i+1)$ respectively: we are heading for a definition of the form $map{\cdot}f{\cdot}(a\,;x) = E$; therefore, we try to derive, for $map{\cdot}f{\cdot}(a\,;x){\cdot}i$, a formula of the form $E{\cdot}i$. The reintroduction of the operator $;$ serves to satisfy the requirement that expression E be a list: by induction hypothesis, $map{\cdot}f{\cdot}x$ is a list of size $\#x$; hence, $f{\cdot}a \,;\, map{\cdot}f{\cdot}x$ is a list of size $\#(a;x)$. From this derivation, we conclude that (0c) is satisfied by $map{\cdot}f{\cdot}(a;x) = f{\cdot}a \,;\, map{\cdot}f{\cdot}x$. Thus, we obtain the following definition:

$$\begin{aligned} & map{\cdot}f{\cdot}[\,] &&= [\,] \\ \& \; & map{\cdot}f{\cdot}(a;x) &&= f{\cdot}a \;;\; map{\cdot}f{\cdot}x \quad . \end{aligned}$$

This definition satisfies the specification for infinite lists too. For infinite lists, formula (0c) , with the universal quantification over x made explicit, can be rewritten to $(\mathbf{A}i::(\mathbf{A}x::map{\cdot}f{\cdot}x{\cdot}i = f{\cdot}(x{\cdot}i)))$, which lends itself to proof by induction over i . Observing that, in the above derivation for $map{\cdot}f{\cdot}(a;x){\cdot}(i+1)$, the step with hint "induction hypothesis" pertains to $map{\cdot}f{\cdot}x{\cdot}i$, with $i < i+1$, we conclude that the above derivation remains valid for infinite x . The only part in the above derivation not applicable to infinite lists is the conclusion that $f{\cdot}a \,;\, map{\cdot}f{\cdot}x$ is a list of the proper size. For this part, however, it suffices to observe that map's definition is uniformly productive (cf. section 5.5.5); hence $map{\cdot}f$ has type $L_\infty \rightarrow L_\infty$.

The operation of applying a function to all elements of a list occurs so often that we introduce a binary operator for it. Because of the resemblance of this operator with function composition, we use the same symbol as for function composition. Thus, we write $f \circ x$ instead of $map{\cdot}f{\cdot}x$. The ambiguity caused by the assignment of two meanings to symbol $\circ$ must be resolved in the context in which it is used. As a result, operator $\circ$ is associative.

definition 6.3.0: The binary operator $\circ$ ("map") is defined as follows. For function f and list x :

$$L{\cdot}(f{\circ}x) \;\wedge\; \#(f{\circ}x) = \#x \;\wedge\; (\mathbf{A}i : 0 \leq i < \#x : (f{\circ}x){\cdot}i = f{\cdot}(x{\cdot}i)) \quad .$$

□

property 6.3.1: Functions f and g and list x satisfy:

$$f{\circ}(g{\circ}x) \;=\; (f{\circ}g){\circ}x$$

□

Similarly, we introduce an operator $\odot$ ("zap") to apply a, sufficiently long, list of functions element by element to a list.

definition 6.3.2: The binary operator $\odot$ ("zap") is defined as follows. For list fs of functions and list x , such that $\#x \leqslant \#fs$:

$$L\cdot(fs \odot x) \wedge \#(fs \odot x) = \#x \wedge (\mathbf{A}\, i : 0 \leqslant i < \#x : (fs \odot x)\cdot i = (fs\cdot i)\cdot(x\cdot i)\,) \quad .$$

□

Syntactically, we assume that $\circ$ and $\odot$ are left-binding and that they bind stronger than ; and $+\!\!+$. The following property shows that $\odot$ is more general than $\circ$.

property 6.3.3: For function f and list x we have:

$$f \circ x \;=\; fs \odot x \;[\![\; fs = f\,;\,fs \;]\!] \quad .$$

□

postulate 6.3.4 (time complexity of $\circ$ and $\odot$): The time complexities of $\circ$ and $\odot$ are linear in the following meaning of the word: the time needed to evaluate $(f\circ x)\!\uparrow\! i$ and $(fs \odot x)\!\uparrow\! i$ is $O(i)$ plus, of course, the time needed to evaluate the applications $f\cdot(x\cdot j)$ or $(fs\cdot j)\cdot(x\cdot j)$, for all j , $0 \leqslant j < i$.

□

example 6.3.5: Function sqr from example 5.9.3 can be defined by means of $\circ$ as follows: $sqr\cdot x \;=\; sq \circ x \;[\![\; sq\cdot a = a \star a \;]\!]$.

□

example 6.3.6: With sum for the function yielding the sum of the elements of a finite integer list, we have, for any finite list x :

$$\#x \;=\; sum\cdot(one \circ x) \;\; [\![\; one\cdot a = 1 \;]\!] \quad .$$

□

example 6.3.7: For function f and lists x , y and z , of equal length, the list whose i-th element is $f\cdot(x\cdot i)\cdot(y\cdot i)\cdot(z\cdot i)$ is $f \circ x \odot y \odot z$. Similarly, $(+) \circ x \odot y$ is the list of sums of the corresponding elements of x and y .

□

example 6.3.8: With s for the infinite list of Fibonacci numbers (see example 5.5.7.6), we have: $s \cdot 0 = 0 \wedge s \cdot 1 = 1$ and, for $i : 0 \leqslant i$, $s \cdot (i+2) = s \cdot (i+1) + s \cdot i$; i.e, $(s{\downarrow}2) \cdot i = (s{\downarrow}1) \cdot i + s \cdot i$. Therefore, s can be defined by:

$s \;=\; 0 \;;\; 1 \;;\; (+) \circ (s{\downarrow}1) \circledcirc s$, or, equivalently,

$s \;=\; 0 \;;\; t \;[\![\; t = 1 \;;\; (+) \circ t \circledcirc s \;]\!]$.

Using postulates 5.10.1 and 6.3.4, we conclude that s has linear time complexity.

□

6.4 List representation of functions

In this section we study the following problem. For given function F , with domain Nat , we wish to derive a definition for an infinite list x representing F , i.e. x's specification is:

(0a) $\mathcal{L}_\infty \cdot x$

(0b) $(\mathrm{A}\, i : 0 \leqslant i : x \cdot i = F \cdot i)$.

We do so by induction on i :

$x \cdot 0$

= { (0b) }

$F \cdot 0$

= { definition of ; }

$(F \cdot 0 \;;\; ?) \cdot 0$,

and, for $i : 0 \leqslant i$:

$x \cdot (i+1)$

= { (0b) }

$F \cdot (i+1)$

= { heading for a formula ending in $\cdot i$ }

$F\cdot((+1)\cdot i)$

$=$ { definition of $\circ$ (function composition) }

$(F\circ(+1))\cdot i$.

The latter formula is not a recursive instance of x's specification: instead of F , it contains $F\circ(+1)$. Both $F\circ(+1)$ and F are -- generalisation by abstraction -- instances of a more general kind of expression: both are, because $F = F\circ(+0)$, of the form $F\circ(+j)$, which can be represented by a parameter j , $0\leq j$. Thus, we obtain $x = y\cdot 0$, where y is specified by:

(1a) $(\mathbf{A}\, j : 0\leq j : L_\infty\cdot(y\cdot j))$

(1b) $(\mathbf{A}\, i,j : 0\leq i \wedge 0\leq j : y\cdot j\cdot i = F\cdot(j+i))$.

By redoing the above derivations, we obtain $y\cdot j\cdot 0 = (F\cdot j\, ;\, ?)\cdot 0$, and:

$y\cdot j\cdot(i+1)$

$=$ { (1b) }

$F\cdot(j+i+1)$

$=$ { algebra }

$F\cdot(j+1+i)$

$=$ { induction hypothesis: (1b) }

$y\cdot(j+1)\cdot i$

$=$ { definition of ; }

$(?\ ;\ y\cdot(j+1))\cdot(i+1)$.

Combination of these results yields the following definitions for x and y :

(2) $x = y\cdot 0\ \llbracket\ y\cdot j = F\cdot j\ ;\ y\cdot(j+1)\ \rrbracket$.

Thus, for each expression F , representing a function on Nat , an infinite-list expression representing that function can be constructed. Similarly, if we need a finite list s , of length n , such that $(\mathbf{A}\, i : 0\leq i<n : s\cdot i = F\cdot i)$, we can derive a definition for s . We can, however, also use infinite list x , specified by (0) , for this purpose: we simply take $x{\uparrow}n$ for s .

6.5 List representation of sets

In this and the next two sections, we study representations of infinite subsets of Nat by means of infinite lists and manipulations thereof. For this purpose, we introduce some additional notions. The element type of sets and lists in this discussion is Nat ; this is left implicit. Throughout this and the next sections, a,b,i,j have type Nat , s,t have type $\mathcal{L}$(Nat) , and x,y,z have type $\mathcal{L}_{\infty}$(Nat) .

definition 6.5.0 (set abstraction): For list s , ⟦s⟧ ("set s") is defined by:

$$[\![s]\!] = \{\, s{\cdot}i \mid i < \#s \,\} \quad .$$

□

Sets are represented by lists by enumeration of their elements. Thus, the set represented by list s is ⟦s⟧ . From this definition, the following properties follow immediately.

property 6.5.1:

$$[\![[\,]]\!] = \emptyset$$
$$[\![a ; s]\!] = \{a\} \cup [\![s]\!]$$

□

For reasons of efficiency, we restrict the use of this representation to *increasing* lists.

definition 6.5.2: Predicate inc is defined, on $\mathcal{L}$, by:

$$inc{\cdot}s \equiv (\mathbf{A}\, i,j : i<j<\#s : s{\cdot}i < s{\cdot}j\,)$$

□

property 6.5.3: For infinite list x , we have:

$$inc{\cdot}x \equiv (\mathbf{A}\, i :: inc{\cdot}(x{\uparrow}i)\,)$$

□

corollary 6.5.4: According to lemma 5.5.4.1, predicate inc is admissible.

□

property 6.5.5:

$\text{inc}\cdot[\,]$

$\text{inc}\cdot(a\,;s) \;\equiv\; (\mathbf{A}\, i : i<\#s : a<s\cdot i\,) \wedge \text{inc}\cdot s$, or, using $[\![\cdot]\!]$:

$\text{inc}\cdot(a\,;s) \;\equiv\; (\mathbf{A}\, b : [\![s]\!]\cdot b : a<b\,) \wedge \text{inc}\cdot s$

□

We now consider predicates P , on L_∞ , of the following form:

(0) $\quad P\cdot x \;\equiv\; \text{inc}\cdot x \wedge [\![x]\!] = V$,

for some, fixed, subset V of Nat . Such predicates are admissible. The proof of this requires the introduction of an operator ↑ ("take"), taking subsets of Nat and naturals for its arguments, defined as follows. For nonempty V , we denote its minimum by min·V .

definition 6.5.6 (the operator ↑ for sets): For set V and natural i :

$V{\uparrow}0 \;=\; \emptyset$

$\emptyset{\uparrow}i \;=\; \emptyset$

$V{\uparrow}(i+1) \;=\; \{\text{min}\cdot V\} \cup (V\backslash\{\text{min}\cdot V\}){\uparrow}i$, for nonempty V .

In words: $V{\uparrow}i$ is the set of the smallest $i\,\textbf{min}\,\#V$ elements of V .

□

property 6.5.7: For set V , we have: $V = (\bigcup i :: V{\uparrow}i\,)$.

□

property 6.5.8: For list s , we have: $\text{inc}\cdot s \Rightarrow (\mathbf{A}\, i :: [\![s]\!]{\uparrow}i = [\![s{\uparrow}i]\!]\,)$.

□

property 6.5.9: For list s , $s\neq[\,]$, we have:

$\text{inc}\cdot s \;\Rightarrow\; s\cdot 0 = \text{min}\cdot[\![s]\!]$

$\text{inc}\cdot s \;\Rightarrow\; [\![s{\downarrow}1]\!] = [\![s]\!]\backslash\{\text{min}\cdot[\![s]\!]\}$

□

property 6.5.10: For increasing lists s and t , we have:

$[\![s]\!] = [\![t]\!] \;\equiv\; s=t$

□

property 6.5.11: Predicate P , as defined by (0) , satisfies:

$P\cdot x \;\equiv\; (\mathbf{A}\, i :: \text{inc}\cdot(x{\uparrow}i) \wedge [\![x{\uparrow}i]\!] = V{\uparrow}i\,)$

□

corollary 6.5.12: P is admissible.
□

We now derive a definition of a function L that maps an infinite subset of Nat on its representation by an increasing infinite list. I.e, L's specification is, for infinite set V :

(1a) $\mathcal{L}_\infty \cdot (L \cdot V)$
(1b) $\text{inc} \cdot (L \cdot V) \wedge [\![L \cdot V]\!] = V$.

Part (1a) will be satisfied by an appeal to the uniform-productivity theorem. I.e, we investigate the feasability of a definition of the following form:

(2) $L \cdot V = a ; x$.

By calculation, we derive for what a and x this definition satisfies (1b) :

$$\begin{array}{cl}
 & \text{inc} \cdot (a ; x) \wedge [\![a ; x]\!] = V \\
= & \quad \{ \text{ property 6.5.5 , property 6.5.1 } \} \\
 & (\mathbf{A} b : [\![x]\!] \cdot b : a < b) \wedge \text{inc} \cdot x \wedge (\{a\} \cup [\![x]\!]) = V \\
\Leftarrow & \quad \{ \text{ set calculus } \} \\
 & (\mathbf{A} b : [\![x]\!] \cdot b : a < b) \wedge \text{inc} \cdot x \wedge V \cdot a \wedge [\![x]\!] = V \backslash \{a\} \\
= & \quad \{ \text{ Leibniz , rearranging terms } \} \\
 & V \cdot a \wedge (\mathbf{A} b : (V \backslash \{a\}) \cdot b : a < b) \wedge \text{inc} \cdot x \wedge [\![x]\!] = V \backslash \{a\} \\
= & \quad \{ \text{ definition of minimum } \} \\
 & a = \text{min} \cdot V \wedge \text{inc} \cdot x \wedge [\![x]\!] = V \backslash \{a\} \\
\Leftarrow & \quad \{ V \backslash \{a\} \text{ is infinite too; specification of } L \text{ (1b) } \} \\
 & a = \text{min} \cdot V \wedge x = L \cdot (V \backslash \{a\}) \quad .
\end{array}$$

The appeal, in the last step, to the specification of L in order to obtain a recursive application of L , is correct because (1b) is admissible. Substitution of the expressions thus obtained for a and x into (2) yields the following definition for L :

(3) $L \cdot V = \text{min} \cdot V \ ; \ L \cdot (V \backslash \{ \text{min} \cdot V \})$.

Notice that the correctness of (the derivation of) this definition crucially depends on the properties of Nat that every nonempty subset has a minimum element and that every subset V equals $(\cup i :: V{\uparrow}i)$. In definition (3) , we have used set operations that are not part of our program notation. In applications of (3) , these set operations must, therefore, be replaced by representations thereof.

6.6 Merge

In this section we use the results from the previous section to derive a definition of a function mrg ("merge") with specification -- remember that x and y denote infinite lists -- :

$$inc{\cdot}x \wedge inc{\cdot}y \Rightarrow mrg{\cdot}x{\cdot}y = z\,[\![\, z : L_\infty{\cdot}z \wedge inc{\cdot}z \wedge [\![z]\!] = [\![x]\!] \cup [\![y]\!] \,]\!] \quad .$$

In words, $mrg{\cdot}x{\cdot}y$ represents the union of the sets represented by x and y . Using function L from the previous section, we rewrite this specification as follows:

$$inc{\cdot}x \wedge inc{\cdot}y \Rightarrow mrg{\cdot}x{\cdot}y = L{\cdot}([\![x]\!] \cup [\![y]\!]) \quad .$$

This specification shows that the pair of lists x,y can considered to represent a set, namely $[\![x]\!] \cup [\![y]\!]$. For the sake of clarity, we introduce a name for the function mapping x,y to $[\![x]\!] \cup [\![y]\!]$, giving:

$$inc{\cdot}x \wedge inc{\cdot}y \Rightarrow mrg{\cdot}x{\cdot}y = L{\cdot}(M{\cdot}x{\cdot}y)$$

(4) $\quad M{\cdot}x{\cdot}y = [\![x]\!] \cup [\![y]\!] \quad .$

In order to be able to apply (3) , we must derive expressions for $min{\cdot}(M{\cdot}x{\cdot}y)$ and for $M{\cdot}x{\cdot}y \backslash \{min{\cdot}(M{\cdot}x{\cdot}y)\}$, for increasing x and y :

$$\begin{array}{ll} & min{\cdot}(M{\cdot}x{\cdot}y) \\ = & \quad \{ (4) \} \\ & min{\cdot}([\![x]\!] \cup [\![y]\!]) \\ = & \quad \{ \text{min distributes over } \cup \} \end{array}$$

$\quad \min\cdot[\![x]\!]\ \mathbf{min}\ \min\cdot[\![y]\!]$
$= \quad \{\ inc\cdot x \wedge inc\cdot y$: property 6.5.9 $\}$
$\quad x\cdot 0\ \mathbf{min}\ y\cdot 0$.

With a as abbreviation for $x\cdot 0\ \mathbf{min}\ y\cdot 0$, we derive:

$\quad \mathfrak{m}\cdot x\cdot y \backslash \{\min\cdot(\mathfrak{m}\cdot x\cdot y)\}$
$= \quad \{$ see above $\}$
$\quad \mathfrak{m}\cdot x\cdot y \backslash \{a\}$
$= \quad \{\ (4)\ \}$
$\quad ([\![x]\!] \cup [\![y]\!]) \backslash \{a\}$
$= \quad \{\ \backslash$ distributes over $\cup\ \}$
$\quad [\![x]\!] \backslash \{a\} \cup [\![y]\!] \backslash \{a\}$.

This formula is symmetric in x and y ; hence, we can continue this derivation with one half of the formula only; we distinguish two cases:

case $x\cdot 0 \leqslant y\cdot 0$:
$\quad [\![x]\!] \backslash \{a\}$
$= \quad \{\ x\cdot 0 \leqslant y\cdot 0 \Rightarrow a = x\cdot 0\ \}$
$\quad [\![x]\!] \backslash \{x\cdot 0\}$
$= \quad \{\ x = x\cdot 0 ; x\downarrow 1$: property 6.5.1 $\}$
$\quad (\{x\cdot 0\} \cup [\![x\downarrow 1]\!]) \backslash \{x\cdot 0\}$
$= \quad \{\ \backslash$ distributes over $\cup$, $inc\cdot x$, hence $\neg([\![x\downarrow 1]\!]\cdot(x\cdot 0))\ \}$
$\quad [\![x\downarrow 1]\!]$,

case $y\cdot 0 < x\cdot 0$:
$\quad [\![x]\!] \backslash \{a\}$
$= \quad \{\ y\cdot 0 < x\cdot 0 \Rightarrow a < x\cdot 0$, $a < x\cdot 0 \wedge inc\cdot x \Rightarrow \neg([\![x]\!]\cdot a)\ \}$
$\quad [\![x]\!]$.

Combination of these cases with the cases for y , elimination of name a , and application of (4) to replace $[\![x\downarrow 1]\!] \cup [\![y]\!]$ by $\mathfrak{m}\cdot(x\downarrow 1)\cdot y$, et cetera, yields:

$$\begin{array}{lll} M\cdot x\cdot y \backslash \{\min\cdot(M\cdot x\cdot y)\} = & (\ x\cdot 0 < y\cdot 0 & \rightarrow M\cdot(x{\downarrow}1)\cdot y \\ & [\!]\ x\cdot 0 = y\cdot 0 & \rightarrow M\cdot(x{\downarrow}1)\cdot(y{\downarrow}1) \\ & [\!]\ x\cdot 0 > y\cdot 0 & \rightarrow M\cdot x\cdot(y{\downarrow}1) \\ &) & . \end{array}$$

By plugging these expressions into (3) , we conclude that L and M satisfy:

$$\begin{array}{lll} L\cdot(M\cdot x\cdot y) = & (\ x\cdot 0 < y\cdot 0 & \rightarrow x\cdot 0\ ;\ L\cdot(M\cdot(x{\downarrow}1)\cdot y) \\ & [\!]\ x\cdot 0 = y\cdot 0 & \rightarrow x\cdot 0\ ;\ L\cdot(M\cdot(x{\downarrow}1)\cdot(y{\downarrow}1)) \\ & [\!]\ x\cdot 0 > y\cdot 0 & \rightarrow y\cdot 0\ ;\ L\cdot(M\cdot x\cdot(y{\downarrow}1)) \\ &) & . \end{array}$$

This formula inspires us to consider the following definition of mrg :

$$\begin{array}{lll} mrg\cdot x\cdot y = & (\ x\cdot 0 < y\cdot 0 & \rightarrow x\cdot 0\ ;\ mrg\cdot(x{\downarrow}1)\cdot y \\ & [\!]\ x\cdot 0 = y\cdot 0 & \rightarrow x\cdot 0\ ;\ mrg\cdot(x{\downarrow}1)\cdot(y{\downarrow}1) \\ & [\!]\ x\cdot 0 > y\cdot 0 & \rightarrow y\cdot 0\ ;\ mrg\cdot x\cdot(y{\downarrow}1) \\ &) & . \end{array}$$

Because, apparently, $mrg\cdot x\cdot y$ and $L\cdot(M\cdot x\cdot y)$ are solutions to the same defining equation, and because this defining equation is productive, we conclude that, for increasing x and y , we have $mrg\cdot x\cdot y = L\cdot(M\cdot x\cdot y)$.

6.7 Filter

As an example of a non-uniformly productive definition, we study the function flt ("filter"), of type $L_\infty \rightarrow L_\infty$, with the following specification:

$$inc\cdot x \Rightarrow inc\cdot(flt\cdot x) \wedge [\![flt\cdot x]\!] = \{ x\cdot i \mid 0 \leqslant i \wedge p\cdot(x\cdot i) \} \quad .$$

Here, p is assumed to be a known function having type Nat→Bool . Notice that the consequent of this specification is an admissible predicate. In order that $flt\cdot x$ be an infinite list the set $\{ x\cdot i \mid 0 \leqslant i \wedge p\cdot(x\cdot i) \}$ must be infinite; i.e. x must contain infinitely many elements satisfying p . Formally, this requirement amounts to $Q\cdot x$, with Q defined by:

$$Q\cdot x \;\equiv\; (\mathbf{A}\, i :: (\mathbf{E}\, j : i \leq j : p\cdot(x\cdot j)))\quad .$$

So, we strengthen the antecedent of the above specification to $inc\cdot x \wedge Q\cdot x$.

In this case, no simple expression exists for the minimum of the set $\{x\cdot i \mid 0 \leq i \wedge p\cdot(x\cdot i)\}$; therefore, we take a slightly different approach:

$\{x\cdot i \mid 0 \leq i \wedge p\cdot(x\cdot i)\}$

$=$ { range split }

$\{x\cdot 0 \mid p\cdot(x\cdot 0)\} \cup \{x\cdot(i+1) \mid 0 \leq i \wedge p\cdot(x\cdot(i+1))\}$

$=$ { $x\cdot(i+1) = (x{\downarrow}1)\cdot i$ }

$\{x\cdot 0 \mid p\cdot(x\cdot 0)\} \cup \{(x{\downarrow}1)\cdot i \mid 0 \leq i \wedge p\cdot((x{\downarrow}1)\cdot i)\}$

$=$ { specification of flt $(inc\cdot x \wedge Q\cdot x \Rightarrow inc\cdot(x{\downarrow}1) \wedge Q\cdot(x{\downarrow}1))$ }

$\{x\cdot 0 \mid p\cdot(x\cdot 0)\} \cup [\![flt\cdot(x{\downarrow}1)]\!]$

$=$ { case analysis }

```
(   p·(x·0) → {x·0} ∪ [[ flt·(x↓1) ]]
[] ¬p·(x·0) →          [[ flt·(x↓1) ]]
)
```

$=$ { property 6.5.1 }

```
(   p·(x·0) → [[ x·0 ; flt·(x↓1) ]]
[] ¬p·(x·0) →       [[ flt·(x↓1) ]]
)    .
```

Because $x\cdot 0 = min\cdot[\![x]\!]$ and because $\{x\cdot i \mid 0 \leq i \wedge p\cdot(x\cdot i)\}$ is a subset of $[\![x]\!]$, $x\cdot 0$ is, in the case $p\cdot(x\cdot 0)$, also the minimum of $\{x\cdot i \mid 0 \leq i \wedge p\cdot(x\cdot i)\}$. This is sufficient to guarantee that $x\cdot 0 \,;\, flt\cdot(x{\downarrow}1)$ is increasing. As a result, we obtain the following recursive definition for flt :

```
flt·(a ; x)  =  (   p·a → a ; flt·x
                [] ¬p·a →     flt·x
                )    .
```

This definition is non-uniformly productive. Requirements (12b) , (12c) , (13) from section 5.5.6 are met, because we have $inc\cdot x \wedge Q\cdot x \Rightarrow inc\cdot(x{\downarrow}1) \wedge Q\cdot(x{\downarrow}1)$ and $inc\cdot x \wedge Q\cdot x \Rightarrow (\mathbf{E}\, i :: p\cdot((x{\downarrow}i)\cdot 0))$.

6.8 Minsegsum

In this section, we derive a program for the minimal sum over all *segments* -- see also section 5.11 -- of an integer list. Formally, we are interested in function f specified by:

$$f\cdot x \;=\; (\mathbf{MIN}\; s,t,u : x = s +\!\!+\, t +\!\!+\, u : sum\cdot t) \quad , \text{ for finite integer list } x \;.$$

Here, sum is a function yielding the sum of the elements of the list supplied as its argument. A recursive definition of sum is:

(0) $sum\cdot[\,] = 0$

(1) & $sum\cdot(a\,;s) = a + sum\cdot s$.

Throughout this section, a and b denote integers, whereas x, s, t, u denote (finite) integer lists. For f , we derive:

$f\cdot[\,]$

= { specification of f }

$(\mathbf{MIN}\; s,t,u : [\,] = s +\!\!+\, t +\!\!+\, u : sum\cdot t)$

= { $+\!\!+$ property (5.8.1) (applied twice) }

$(\mathbf{MIN}\; s,t,u : s=[\,] \wedge t=[\,] \wedge u=[\,] : sum\cdot t)$

= { one-point rule }

$sum\cdot[\,]$

= { (0) }

0 .

Furthermore:

$f\cdot(a\,;x)$

= { specification of f }

$(\mathbf{MIN}\; s,t,u : a\,;x = s +\!\!+\, t +\!\!+\, u : sum\cdot t)$

= { range split, splitting cases $s=[\,]$ and $s\neq[\,]$ }

$(\mathbf{MIN}\; s,t,u : s=[\,] \wedge a\,;x = s +\!\!+\, t +\!\!+\, u : sum\cdot t)$ **min**
$(\mathbf{MIN}\; s,t,u : s\neq[\,] \wedge a\,;x = s +\!\!+\, t +\!\!+\, u : sum\cdot t)$

= { one-point rule and $[\,] +\!\!+\, t = t$, dummy substitution $s \leftarrow b\,;s$ }

$(\mathbf{MIN}\, t,u : a\,;x = t \mathbin{+\!\!+} u : sum\cdot t)$ **min** $(\mathbf{MIN}\, b,s,t,u : a\,;x = b\,;s \mathbin{+\!\!+} t \mathbin{+\!\!+} u : sum\cdot t)$

$=$ { introduction of function g (see below) , ; property (5.8.0) }

$g\cdot(a\,;x)$ **min** $(\mathbf{MIN}\, b,s,t,u : a = b \wedge x = s \mathbin{+\!\!+} t \mathbin{+\!\!+} u : sum\cdot t)$

$=$ { one-point rule }

$g\cdot(a\,;x)$ **min** $(\mathbf{MIN}\, s,t,u : x = s \mathbin{+\!\!+} t \mathbin{+\!\!+} u : sum\cdot t)$

$=$ { induction hypothesis }

$g\cdot(a\,;x)$ **min** $f\cdot x$.

The specification of function g follows directly from the way it has been used in the above derivation:

$g\cdot x = (\mathbf{MIN}\, t,u : x = t \mathbin{+\!\!+} u : sum\cdot t)$, for finite integer list x .

The specification of g is simpler than the specification of f ; in very much the same way as above, we can derive $g\cdot[\,] = 0$ and:

$g\cdot(a\,;x)$

$=$ { specification of g }

$(\mathbf{MIN}\, t,u : a\,;x = t \mathbin{+\!\!+} u : sum\cdot t)$

$=$ { range split: $t=[\,] \vee t \neq [\,]$ and subsequent simplification }

$sum\cdot[\,]$ **min** $(\mathbf{MIN}\, t,u : x = t \mathbin{+\!\!+} u : sum\cdot(a\,;t))$

$=$ { (0) and (1) }

0 **min** $(\mathbf{MIN}\, t,u : x = t \mathbin{+\!\!+} u : a + sum\cdot t)$

$=$ { + distributes over **MIN** ; induction hypothesis for g }

0 **min** $(a + g\cdot x)$.

Thus, we have derived the following definitions for f and g :

$f\cdot[\,] = 0$
& $f\cdot(a\,;x) = g\cdot(a\,;x)$ **min** $f\cdot x$
& $g\cdot[\,] = 0$
& $g\cdot(a\,;x) = 0$ **min** $(a + g\cdot x)$.

By unfolding $g\cdot(a;x)$ and using that $0\ \mathbf{min}\ f\cdot x = f\cdot x$, because $f\cdot x \leqslant 0$, we can replace the second line of this definition by:

$$\& \ f\cdot(a;x) \ = \ (a+g\cdot x)\ \mathbf{min}\ f\cdot x \qquad .$$

By means of tupling, we transform this definition into the following, more efficient one -- with $h\cdot x = [f\cdot x , g\cdot x]$ -- :

$$\begin{array}{lll} f\cdot x \ = \ h\cdot x\cdot 0 \ [\![& h\cdot[\,] & = [0,0] \\ & \& \ h\cdot(a;x) & = [\,(a+c)\ \mathbf{min}\ b , 0\ \mathbf{min}\ (a+c)\,] \ [\![\ [b,c] = h\cdot x \ ]\!] \\ &]\!] \quad . & \end{array}$$

The time complexity of $f\cdot x$ now is $O(\#x)$. By means of the Tail Recursion Theorem for lists (6.2.0), the definition of h can be transformed into a tail recursive one: take $[0,0]$ for X and define $\oplus$ by

$$a \oplus [b,c] \ = \ [\,(a+c)\ \mathbf{min}\ b , 0\ \mathbf{min}\ (a+c)\,] \qquad .$$

Notice that from f's specification it follows that $f\cdot(rev\cdot x) = f\cdot x$, because $sum\cdot(rev\cdot x) = sum\cdot x$. Thus, we obtain the following definition for f :

$$\begin{array}{lll} f\cdot x \ = \ G\cdot[0,0]\cdot x\cdot 0 \ [\![& G\cdot y\cdot[\,] & = y \\ & \& \ G\cdot[b,c]\cdot(a;x) & = G\cdot[\,(a+c)\ \mathbf{min}\ b , 0\ \mathbf{min}\ (a+c)\,]\cdot x \\ &]\!] \quad . & \end{array}$$

6.9 Carré recognition

As an exercise in the use of ↑↓-calculus, we consider the following problem. A finite list is called a *carré* if it equals $s \mathbin{+\!\!+} s$, for some finite list s . Consequently, carrés always have even lengths. We wish to derive a definition for function f , having type $L_\infty(\mathrm{Int}) \rightarrow L_\infty(\mathrm{Bool})$, with the following informal specification:

$$f\cdot x\cdot i \ \equiv \ \text{"}x\!\uparrow\!(2*i) \text{ is a carré"} \ , \ 0 \leqslant i \qquad .$$

This specification can be formalised as follows. A finite list s of length $2*i$ is a carré if $s\uparrow i = s\downarrow i$, because $s = s\uparrow i +\!\!+ s\downarrow i$ and $\#(s\uparrow i) = \#(s\downarrow i)$. Because $x\uparrow(2*i)\uparrow i = x\uparrow i$ and $x\uparrow(2*i)\downarrow i = x\downarrow i\uparrow i$ (property 5.8.3), f's specification becomes:

$$f{\cdot}x{\cdot}i \;\equiv\; x\uparrow i = x\downarrow i\uparrow i \quad , 0 \leqslant i \quad .$$

We derive, using induction on i :

$$\begin{array}{ll} & f{\cdot}x{\cdot}0 \\ = & \quad \{\text{ specification of f }\} \\ & x\uparrow 0 = x\downarrow 0\uparrow 0 \\ = & \quad \{\ x\uparrow 0 = [\,] \text{ and } x\downarrow 0\uparrow 0 = [\,]\ \} \\ & \text{true} \quad , \end{array}$$

and:

$$\begin{array}{ll} & f{\cdot}x{\cdot}(i+1) \\ = & \quad \{\text{ specification of f }\} \\ & x\uparrow(i+1) = x\downarrow(i+1)\uparrow(i+1) \\ = & \quad \{\ x = x{\cdot}0\,;x\downarrow 1 \text{ , hence } x\uparrow(i+1) = x{\cdot}0\,;x\downarrow 1\uparrow i \text{ ; idem for } x\downarrow(i+1)\ \} \\ & x{\cdot}0\,;x\downarrow 1\uparrow i \;=\; (x\downarrow(i+1)){\cdot}0\,;x\downarrow(i+1)\downarrow 1\uparrow i \\ = & \quad \{\ a\,;x = b\,;y \;\equiv\; a = b \wedge x = y\ \} \\ & x{\cdot}0 = (x\downarrow(i+1)){\cdot}0 \;\wedge\; x\downarrow 1\uparrow i = x\downarrow(i+1)\downarrow 1\uparrow i \\ = & \quad \{\ (x\downarrow(i+1)){\cdot}0 = x{\cdot}(i+1) \text{ , } x\downarrow(i+1)\downarrow 1 = x\downarrow 2\downarrow i\ \} \\ & x{\cdot}0 = x{\cdot}(i+1) \;\wedge\; x\downarrow 1\uparrow i = x\downarrow 2\downarrow i\uparrow i \quad . \end{array}$$

The formula $x\downarrow 1\uparrow i = x\downarrow 2\downarrow i\uparrow i$ is not an instance of f's specification. Generalisation by abstraction inspires us to consider function g with specification:

$$g{\cdot}x{\cdot}y{\cdot}i \;\equiv\; x\uparrow i = y\downarrow i\uparrow i \quad , 0 \leqslant i \quad .$$

We then have $f{\cdot}x = g{\cdot}x{\cdot}x$. By adaptation of the above derivations to the more general situation, we obtain:

$g \cdot x \cdot y \cdot 0 \;=\; true$, and:

$g \cdot x \cdot y \cdot (i+1)$

$=$ { as before }

$x \cdot 0 = y \cdot (i+1) \wedge x{\downarrow}1{\uparrow}i = y{\downarrow}2{\downarrow}i{\uparrow}i$

$=$ { induction hypothesis }

$x \cdot 0 = y \cdot (i+1) \wedge g \cdot (x{\downarrow}1) \cdot (y{\downarrow}2) \cdot i$

$=$ { $y \cdot (i+1) = (y{\downarrow}1) \cdot i$, introduction of function h (see below) }

$h \cdot (y{\downarrow}1) \cdot (g \cdot (x{\downarrow}1) \cdot (y{\downarrow}2)) \cdot i$.

Using the definition of ; , we obtain the following definition for g :

$$g \cdot x \cdot y \;=\; true \;;\; h \cdot (y{\downarrow}1) \cdot (g \cdot (x{\downarrow}1) \cdot (y{\downarrow}2)) \quad .$$

The specification of function h , of type $L_\infty(\text{Int}) \to L_\infty(\text{Bool}) \to L_\infty(\text{Bool})$, is:

$$h \cdot u \cdot v \cdot i \;\equiv\; x \cdot 0 = u \cdot i \wedge v \cdot i \quad , 0 \leqslant i \quad .$$

Notice that, because of the occurrence of the global constant x , this specification must be understood in the context of g's defining expression. The i-th element of $h \cdot u \cdot v$ depends on the i-th elements of u and v only; hence, h can be defined in terms of u and v by means of the $\circ$ and $\odot$ operators, as follows:

$x \cdot 0 = u \cdot i \wedge v \cdot i$

$=$ { definition of function $(x \cdot 0 =)$ }

$(x \cdot 0 =) \cdot (u \cdot i) \wedge v \cdot i$

$=$ { definition of $\circ$ (6.3.0) applied to function $(x \cdot 0 =)$ }

$((x \cdot 0 =) \circ u) \cdot i \wedge v \cdot i$

$=$ { definition of $\circ$ and $\odot$ (see example 6.3.7) applied to $(\wedge)$ }

$((\wedge) \circ (x \cdot 0 =) \circ u \odot v) \cdot i$.

By plugging this into the definition for g , we obtain:

$$g \cdot x \cdot y \;=\; true \;;\; (\wedge) \circ (x \cdot 0 =) \circ (y{\downarrow}1) \odot g \cdot (x{\downarrow}1) \cdot (y{\downarrow}2) \quad .$$

By means of parameters patterns, this definition can be cleaned up a little:

$$g \cdot (a\,;x) \cdot (b\,;y) \;=\; \text{true} \;;\; (\wedge) \circ (a=) \circ y \odot g \cdot x \cdot (y{\downarrow}1) \qquad .$$

Of course, it is also possible to derive a recursive definition for h without the use of $\circ$ and $\odot$; such a derivation resembles the derivations of the recursive definition of $\circ$ in section 6.3.

The time complexity of $f \cdot x{\uparrow}n$ is $O(n^2)$. In chapter 10 we show that programs with $O(n)$ time complexity are possible too.

6.10 On the efficiency of infinite-list definitions: a case study

Throughout this section, F is a function of type $L_\infty \to L_\infty \to L_\infty$; we assume that $F \cdot x$ is list-productive, for all $x : L_\infty \cdot x$. We consider functions f and g defined by:

$$
\begin{array}{lll}
(0) & f \cdot x & = F \cdot x \cdot (f \cdot x) \\
(1) & g \cdot x & = y \;[\![\; y = F \cdot x \cdot y \;]\!] \qquad .
\end{array}
$$

From the first productivity theorem, it follows that f and g have type $L_\infty \to L_\infty$, and that f and g are functionally equivalent in the sense that $(\text{A}\, x : L_\infty \cdot x : f \cdot x = g \cdot x)$. Thus, (0) and (1) are different definitions for the same function.

We now compare the time complexities of f and g . Let the time needed to evaluate $F \cdot x \cdot y{\uparrow}i$, for arbitrary (infinite lists) x and y , be at most $T \cdot i$. Let $tf \cdot i$ and $tg \cdot i$ denote the times needed to evaluate $f \cdot x{\uparrow}i$ and $g \cdot x{\uparrow}i$. From (0) and (1) , using that $F \cdot x$ is productive, we then find the following recurrence relations for tf and tg :

$$
\begin{array}{lll}
tf \cdot 0 & = 1 & \\
tf \cdot (i+1) & \leqslant 1 + tf \cdot i + T \cdot (i+1) & ,\, 0 \leqslant i \\
tg \cdot 0 & = 1 & \\
tg \cdot (i+1) & \leqslant 2 + T \cdot (i+1) & ,\, 0 \leqslant i \qquad .
\end{array}
$$

The difference between these relations is due to postulate 5.10.1, on account of which y in the expression $F \cdot x \cdot y$ may be considered as a global constant that needs no evaluation; hence, the time needed to evaluate $F \cdot x \cdot y \uparrow (i+1)$ is (at most) $T \cdot (i+1)$.

Thus, we observe that, generally, definition (0) is less efficient than definition (1) . For example, if $T \cdot i = O(i)$ we find $tf \cdot i = O(i^2)$ and $tg \cdot i = O(i)$. We conclude that the definitions of f and g really are different.

The above observations influence the way in which we derive programs. We illustrate this by means of the following example. Function f , having type $\mathcal{L}_\infty(\mathrm{Int}) \to \mathcal{L}_\infty(\mathrm{Int})$, is specified by:

$$f \cdot x \cdot j \;=\; (\mathbf{S}\, i : 0 \leqslant i < j : x \cdot i) \quad , 0 \leqslant j \qquad .$$

By straightforward calculation two sets of recurrence relations can be derived from this specification:

$$
\begin{array}{lll}
(2) & f \cdot x \cdot 0 & = 0 \\
 & f \cdot x \cdot (j+1) & = x \cdot 0 + f \cdot (x \downarrow 1) \cdot j \quad , 0 \leqslant j \\
(3) & f \cdot x \cdot 0 & = 0 \\
 & f \cdot x \cdot (j+1) & = f \cdot x \cdot j + x \cdot j \quad , 0 \leqslant j \qquad .
\end{array}
$$

(2) resembles the recurrence relations that are usual for functions on finite lists: the function's value is expressed in terms of the head and the tail of the parameter. Nevertheless, (3) is to be preferred. Both (2) and (3) can be easily transformed into definitions satisfying f's specification, giving (4) and (5) respectively:

$$
\begin{array}{ll}
(4) & f \cdot x \;=\; 0 \;;\; (x \cdot 0 +) \circ f \cdot (x \downarrow 1) \\
(5) & f \cdot x \;=\; 0 \;;\; (+) \circ f \cdot x \odot x \qquad .
\end{array}
$$

Whichever definition is used, f has quadratic time complexity. The essential difference, however, is that, in (4) , f occurs in its defining expression with an argument, viz. $x \downarrow 1$, different from x , whereas in (5) only $f \cdot x$ occurs in its defining expression. Hence, definition (5) has the same form as (0) , whereas (4) has not. Definition (5) can now be recoded in the same form as (1) , which yields a definition with linear time complexity:

$$f \cdot x \;=\; y \;[\![\; y \;=\; 0 \;;\; (+) \circ y \odot x \;]\!] \qquad .$$

So much for a story that serves to show that, when efficiency is involved, functional programming is a little trickier than one might expect at first sight. This is a direct consequence of the rather complicated -- compared with sequential programming -- computational model underlying the program notation.

6.11 On the introduction of list parameters

In this section, we study variations on the following recursive definition for function f :

$$f \cdot i \; = \; F \cdot (X \cdot i) \cdot (f \cdot (G \cdot i)) \quad .$$

Here, F and G are functions and X is an infinite list. We are interested in $f \cdot i$ only for natural i ; we assume G to have type $Nat \rightarrow Nat$.

Evaluation of $X \cdot i$ requires $O(i)$ time, which may be too inefficient. In this section, we investigate ways to eliminate the expression $X \cdot i$ from the above definition in order to improve its efficiency.

A standard technique for this purpose is to equip f with an additional parameter representing $X \cdot i$. I.e, we introduce function $f1$ with the following specification:

$$f1 \cdot (X \cdot i) \cdot i \; = \; f \cdot i \quad , 0 \leqslant i \quad .$$

This yields the following definition for $f1$:

$$f1 \cdot a \cdot i \; = \; F \cdot a \cdot (f1 \cdot (X \cdot j) \cdot j) \; [\![\; j = G \cdot i \;]\!] \quad .$$

Of course, this transformation only shifts the problem; now we have $X \cdot j$ as a subexpression. Generally, it would be nice if we could exploit the presence of parameter a for the construction of an (efficient) expression for $X \cdot j$ too. Without further knowledge about X , however, we cannot express $X \cdot j$ in terms of $X \cdot i$. Therefore, we investigate a larger class of ways to introduce additional parameters: the parameter needs not be equal to $X \cdot i$, it suffices that $X \cdot i$ can be expressed efficiently in terms of it.

We distinguish 3 cases, depending on the relation between i and j .

- $i \leq j$: Observing that $X{\cdot}i = (X{\downarrow}i){\cdot}0$, and that $X{\downarrow}j = X{\downarrow}i{\downarrow}(j-i)$, we provide f with an additional parameter representing $X{\downarrow}i$. If we call the function thus obtained f2 its specification becomes:

  $f2{\cdot}(X{\downarrow}i){\cdot}i \;=\; f{\cdot}i \quad , 0 \leq i \qquad .$

  Transforming f's definition accordingly yields:

  $f2{\cdot}x{\cdot}i \;=\; F{\cdot}(x{\cdot}0){\cdot}(f2{\cdot}(x{\downarrow}(j-i)){\cdot}j) \;[\![\; j = G{\cdot}i \;]\!] \qquad .$

  The expression $x{\cdot}0$ has time complexity $O(1)$, which is an improvement to the original $O(i)$; the expression $x{\downarrow}(j-i)$ has time complexity $O(j-i)$, which may or may not be an improvement to the original $O(i)$.

- $j \leq i$: Observing that $X{\cdot}i = rev{\cdot}(X{\uparrow}(i+1)){\cdot}0$, and that $rev{\cdot}(X{\uparrow}(j+1)) = rev{\cdot}(X{\uparrow}(i+1)){\downarrow}(i-j)$, we provide f with an additional parameter representing $rev{\cdot}(X{\uparrow}(i+1))$. If we call the function thus obtained f3 its specification becomes:

  $f3{\cdot}(rev{\cdot}(X{\uparrow}(i+1))){\cdot}i \;=\; f{\cdot}i \quad , 0 \leq i \qquad .$

  The corresponding definition for f3 is:

  $f3{\cdot}y{\cdot}i \;=\; F{\cdot}(y{\cdot}0){\cdot}(f3{\cdot}(y{\downarrow}(i-j)){\cdot}j) \;[\![\; j = G{\cdot}i \;]\!] \qquad .$

  As before, the expressions $y{\cdot}0$ and $y{\downarrow}(i-j)$ have time complexity $O(1)$ and $O(i-j)$; this is never worse than the original $O(i)$.

- true : Without a priori knowledge about the relation between i and j , the only thing we can do is to introduce the above case analysis into the program. This brings about a combination of the above two techniques: the *whole* list X can be represented by the pair $[\, rev{\cdot}(X{\uparrow}i) , X{\downarrow}i \,]$ -- or by the pair $[\, rev{\cdot}(X{\uparrow}(i+1)) , X{\downarrow}(i+1) \,]$ -- . Thus, we obtain function f4 with specification:

$$f4 \cdot [rev \cdot (X{\uparrow}i), X{\downarrow}i] \cdot i \;=\; f \cdot i \quad , 0 \leqslant i \quad .$$

The corresponding program fragment is:

$$f4 \cdot [y,x] \cdot i \;=\; F \cdot (x \cdot 0) \cdot (f4 \cdot (shift \cdot (j-i) \cdot [y,x]) \cdot j) \;\; [\![\; j = G \cdot i \;]\!] \quad ,$$

where function shift is specified by:

$$shift \cdot (j-i) \cdot [rev \cdot (x{\uparrow}i), x{\downarrow}i] \;=\; [rev \cdot (x{\uparrow}j), x{\downarrow}j] \quad , 0 \leqslant i \wedge 0 \leqslant j \wedge L_{\infty} \cdot x \quad .$$

A program for shift is:

```
shift·k·[y,x] =  (k ⩽ 0  →  [ y↓(-k) , rev·(y↑(-k)) ++ x ]
                 ▯k ⩾ 0  →  [ rev·(x↑k) ++ y , x↓k ]
                 )   .
```

The time complexity of shift·k·[y,x] is $O(|k|)$.
(end •)

The above program transformations do not suggest such a marked gain in efficiency that they seem to be worth the trouble. Yet, occasionally, they yield an increase of the program's efficiency by an order of magnitude. This is due to the recursive form of the above definitions, which allows the use of *amortized complexity* -- see chapter 7 -- to take into account the contributions of the individual shift-operations. In chapter 10, we shall encounter an application of this.

7 Two-sided list operations

7.0 Introduction

In most functional-program notations the only elementary list operations are ; ("cons"), hd ("head") and tl ("tail"). The reason for this is probably historical (LISP). Operationally speaking, by means of these operations lists can be manipulated at their "left ends" only. This restricted choice of operations allows an efficient implementation: usually, they are assumed to have $O(1)$ time complexity.

In this chapter we show that it is possible to implement a symmetric set of finite list operations efficiently; the set is symmetric in the sense that lists can be manipulated at either end. The operations have $O(1)$ time complexity, provided that we content ourselves with, so-called, *amortized efficiency*, instead of worst-case efficiency.

The idea behind our design is simple and not new [Gri], but, in order to be effective, its elaboration requires some care. The idea is to represent each list by *a pair of lists*: the pair $[x,y]$ represents the list $x \mathbin{+\!\!+} \mathrm{rev}\cdot y$. Thus, each list can be represented in many ways, and it is by judicious exploitation of this freedom that we achieve our goal.

7.1 Amortized complexity

Without pretending generality, we introduce the notion of *amortized complexity* in a form suiting our purpose.

In this section V is a set and f and t are functions of types $V \to V$ and $V \to Nat$ respectively. For v , $V\cdot v$, we interpret $t\cdot v$ as the *cost*, in some meaning of the word, of evaluation of $f\cdot v$. Now suppose that we are interested in a sequence of successive applications of f ; i.e. we define a sequence x as follows:

$V\cdot x_0$ (x_0 is assumed to be known) , and
$x_{i+1} = f\cdot x_i$, $0 \leq i$.

Computation of the first $n+1$ elements of x then costs $(\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i)$. If the value of this expression, as a function of n , is $O(n)$, then we say that the *amortized cost* of each of f's applications (in sequence x) is $O(1)$. Of course, this is so if t is $O(1)$, but this is not necessary: the requirement that $(\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i)$ is $O(n)$ is weaker. The introduction of amortized cost reflects our decision to be interested only in the cumulative cost of a sequence of successive operations.

For the sake of simplicity, it would be nice if we could discuss the amortized cost of f without introduction of sequence x . This can be done as follows. We introduce a function s , of type $V \rightarrow Nat$, and we interpret $s{\cdot}v$ as the amortized cost of evaluation of $f{\cdot}v$. We try to couple s and t in such a way that, for our sequence x , we have:

$$(\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i) \leq (\mathbf{S}\, i : 0 \leq i < n : s{\cdot}x_i) \quad , 0 \leq n \quad .$$

Consequently, if s is $O(1)$ then the cumulative cost of computing the first $n+1$ elements of x indeed is $O(n)$.

The following idea for a suitable coupling is -- as far as we know -- due to Tarjan [Tar]. We design, or invent, a function c , of type $V \rightarrow Nat$, and define s as follows:

$$s{\cdot}v = t{\cdot}v + c{\cdot}(f{\cdot}v) - c{\cdot}v \quad , \text{ for all } v \, , V{\cdot}v \quad .$$

Under the additional assumption $c{\cdot}x_0 = 0$, we derive:

$$
\begin{aligned}
& (\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i) \\
= & \quad \{ \text{"telescope summation"} \} \\
& (\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i + c{\cdot}x_{i+1} - c{\cdot}x_i) - c{\cdot}x_n + c{\cdot}x_0 \\
= & \quad \{ \text{definition of } x_{i+1} \} \\
& (\mathbf{S}\, i : 0 \leq i < n : t{\cdot}x_i + c{\cdot}(f{\cdot}x_i) - c{\cdot}x_i) - c{\cdot}x_n + c{\cdot}x_0 \\
= & \quad \{ \text{definition of } s \} \\
& (\mathbf{S}\, i : 0 \leq i < n : s{\cdot}x_i) - c{\cdot}x_n + c{\cdot}x_0 \\
\leq & \quad \{ c{\cdot}x_n \geq 0 \, , \, c{\cdot}x_0 = 0 \} \\
& (\mathbf{S}\, i : 0 \leq i < n : s{\cdot}x_i) \quad .
\end{aligned}
$$

What does this mean in practice? In order to prove that a function f, with given cost function t, has amortized cost $O(1)$, it suffices to design a natural function c, the so-called *credit function*, satisfying:

$$c{\cdot}x_0 = 0 \quad , \text{ and}$$
$$t{\cdot}v + c{\cdot}(f{\cdot}v) - c{\cdot}v \text{ is, as function of } v, \; O(1) \quad .$$

Herewith, x_0 represents the initial argument -- or, if one takes a less functional point of view, the initial state -- of the computation.

The above remains valid when f represents an element of a whole *class of functions*, each having its own cost function t. In this case, one and the same credit function must satisfy the above requirement for each pair f, t from this class.

7.2 Specifications

The problem to be solved is to implement an extended set of elementary list operations in such a way that the amortized time complexity of each of these operations is $O(1)$. Here, lists are finite lists. For this purpose, L_* will be represented by a set V, say, such that the representation of lists from L_* by elements of V is not unique. The abstraction function mapping V to L_* is denoted by $[\![\cdot]\!]$; i.e. $[\![s]\!]$ is the list represented by s, for s, $V{\cdot}s$.

For the sake of homogeneity, we use functions hd ("head") and tl ("tail") satisfying, for all x,y:

$$hd{\cdot}(x\,;y) \;=\; x$$
$$tl{\cdot}(x\,;y) \;=\; y \quad .$$

Notice that, for non-empty list x, $hd{\cdot}x = x{\cdot}0$ and $tl{\cdot}x = x{\downarrow}1$. In this exercise we use L_* and its associated functions for two purposes, namely to *specify* the new list operations and to *implement* them.

The functions to be implemented are:

; ("left cons")	and	⁏ ("right cons")
lhd ("left head")	and	rhd ("right head")
ltl ("left tail")	and	rtl ("right tail") .

Using $[\![\cdot]\!]$ and the operations on L_*, we specify these functions as follows; for any a and for s, $V\cdot s$:

$$
\begin{array}{lll}
[\![a;s]\!] & = [a] +\!\!+ [\![s]\!] & \\
[\![s;a]\!] & = [\![s]\!] +\!\!+ [a] & \\
\mathrm{lhd}\cdot s & = \mathrm{hd}\cdot[\![s]\!] & ,\ [\![s]\!] \neq [\,] \\
\mathrm{rhd}\cdot s & = \mathrm{hd}\cdot(\mathrm{rev}\cdot[\![s]\!]) & ,\ [\![s]\!] \neq [\,] \\
[\![\mathrm{ltl}\cdot s]\!] & = \mathrm{tl}\cdot[\![s]\!] & ,\ [\![s]\!] \neq [\,] \\
[\![\mathrm{rtl}\cdot s]\!] & = \mathrm{rev}\cdot(\mathrm{tl}\cdot(\mathrm{rev}\cdot[\![s]\!])) & ,\ [\![s]\!] \neq [\,] \quad .
\end{array}
$$

remark 7.2.0: From these specifications, the types of these functions can be derived.

□

Moreover, we need a representation of the empty list; i.e. we must choose a value $[\,]_V$, $V\cdot[\,]_V$, satisfying:

$$[\![\, [\,]_V]\!] = [\,] \quad .$$

Functions $(a;)$ and $(;a)$, for every a, and functions ltl and rtl have type $V \to V$. They will be implemented in such a way that their amortized time complexity is $O(1)$. The functions lhd and rhd do not fit into this pattern: they are functions from V to elements. This is no problem: we shall see to it that lhd and rhd have $O(1)$ (worst-case) time complexity.

7.3 Representation

Our new lists are represented by pairs of old lists; i.e. we choose $V = L_* \times L_*$. For function $[\![\cdot]\!]$ we choose:

$$[\![\, [x,y] \,]\!] = x +\!\!+ \mathrm{rev}\cdot y \quad .$$

This representation leaves us no choice for the definition of $[\,]_V$: the only solution of the equation $[\,] = x +\!\!+ \mathrm{rev}\cdot y$ is $x = [\,] \wedge y = [\,]$; hence:

$$[\,]_V = [\,[\,],[\,]\,] \quad .$$

We now derive a definition for lhd :

$$
\begin{array}{ll}
 & \mathrm{lhd}\cdot[x,y] \\
= & \quad \{ \text{ specification of lhd } \} \\
 & \mathrm{hd}\cdot[\![\, [x,y] \,]\!] \\
= & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & \mathrm{hd}\cdot(x \mathbin{+\!\!+} \mathrm{rev}\cdot y) \\
= & \quad \{ \text{ properties of } +\!\!+ \text{ , definition of hd } \} \\
 & (\ x \neq [\,] \rightarrow \mathrm{hd}\cdot x \\
 & [\!]\ x = [\,] \rightarrow \mathrm{hd}\cdot(\mathrm{rev}\cdot y) \\
 &) \quad .
\end{array}
$$

Evaluation of $\mathrm{hd}\cdot(\mathrm{rev}\cdot y)$ takes $O(\#y)$ time; hence, the definition thus obtained does not have $O(1)$ time complexity. It does, however, have $O(1)$ time complexity in the special case $x\neq[\,] \vee y=[\,]$. We could, therefore, restrict set V to the pairs $[x,y]$ that satisfy $x\neq[\,] \vee y=[\,]$. The conjunction of this restriction and its symmetric counterpart, $y\neq[\,] \vee x=[\,]$, amounts to $x=[\,] \equiv y=[\,]$, which excludes all possible representations of the singleton lists. Hence, the restriction $x\neq[\,] \vee y=[\,]$ is too strong. We weaken it to $x\neq[\,] \vee \#y\leqslant 1$, or, equivalently, $1\leqslant\#x \vee \#y\leqslant 1$. For, y , $\#y\leqslant 1$, we have $\mathrm{rev}\cdot y = y$; thus, we obtain the following definition for lhd :

$$
\begin{array}{lll}
\mathrm{lhd}\cdot[x,y] & = & (\ x \neq [\,] \rightarrow \mathrm{hd}\cdot x \\
 & & [\!]\ x = [\,] \rightarrow \mathrm{hd}\cdot y \\
 & &) \quad .
\end{array}
$$

By a similar calculation for rhd , we conclude that it seems wise to restrict set V to those pairs $[x,y]$ satisfying $1\leqslant\#y \vee \#x\leqslant 1$. Together, these two restrictions define set V : V now is a subset of $L_* \times L_*$. The relation defining this subset, also called the *representation invariant*, is Q , with:

Q: $\quad (1 \leqslant \#x \ \vee \ \#y \leqslant 1) \ \wedge \ (1 \leqslant \#y \ \vee \ \#x \leqslant 1) \quad .$

The definition for rhd then becomes -- notice the symmetry -- :

$$\begin{array}{lll} \mathrm{rhd}\cdot[y,x] & \doteq & (\ x \neq [\,] \rightarrow \mathrm{hd}\cdot x \\ & & [\!]\ x = [\,] \rightarrow \mathrm{hd}\cdot y \\ & &) \quad . \end{array}$$

For the development of definitions for the other functions we shall use the following simple lemma.

lemma 7.3.0: $\#x = 1 \vee \#y = 1 \Rightarrow Q$
□

7.4 Left and right cons

The derivation of definitions for ; and ⁏ is straightforward if we temporarily forget the proof obligation with respect to Q . We perform these derivations in parallel:

$$\begin{array}{ll|ll} & [\![\, a\,;[x,y]\,]\!] & & [\![\,[y,x]\,⁏\,a\,]\!] \\ = & \quad \{\text{ specification of } ; \} & = & \quad \{\text{ specification of } ⁏ \} \\ & [a] +\!\!+ [\![\,[x,y]\,]\!] & & [\![\,[y,x]\,]\!] +\!\!+ [a] \\ = & \quad \{\text{ definition of } [\![\cdot]\!] \} & = & \quad \{\text{ definition of } [\![\cdot]\!] \} \\ & [a] +\!\!+ x +\!\!+ \mathrm{rev}\cdot y & & y +\!\!+ \mathrm{rev}\cdot x +\!\!+ [a] \\ = & \quad \{\text{ list calculus }\} & = & \quad \{\text{ list calculus }\} \\ & (a\,;x) +\!\!+ \mathrm{rev}\cdot y & & y +\!\!+ \mathrm{rev}\cdot(a\,;x) \\ = & \quad \{\text{ definition of } [\![\cdot]\!] \} & = & \quad \{\text{ definition of } [\![\cdot]\!] \} \\ & [\![\,[a\,;x\,,y]\,]\!] \quad . & & [\![\,[y\,,a\,;x]\,]\!] \quad . \end{array}$$

Thus, we conclude that the specifications of ; and ⁏ are satisfied by:

$$\begin{array}{lll} & a\,;[x,y] & = [a\,;x\,,y] \\ \& & [y,x]\,⁏\,a & = [y\,,a\,;x] \quad . \end{array}$$

The expression $[a\,;x\,,y]$, however, need not satisfy Q : Q's second conjunct may be false, but it certainly is true if $1 \leq \#y$. For the special case $y = [\,]$, we redo the above dervation:

$$
\begin{array}{ll}
 & [\![\, a\,;[x,[\,]]\,]\!] \\
= & \quad \{ \text{ as before , with } y \leftarrow [\,] \text{ } \} \\
 & [a] + \!\! + \; x + \!\! + \; \mathrm{rev}\cdot[\,] \\
= & \quad \{ \text{ } \mathrm{rev}\cdot[\,] = [\,] \text{ ; } [\,] \text{ is the identity of } + \!\! + \text{ } \} \\
 & [a] + \!\! + \; x \\
= & \quad \{ \text{ } [x,[\,]] \text{ satisfies } Q \text{ , hence } \#x \leqslant 1 \text{ , hence } x = \mathrm{rev}\cdot x \text{ } \} \\
 & [a] + \!\! + \; \mathrm{rev}\cdot x \\
= & \quad \{ \text{ definition of } [\![\,\cdot\,]\!] \text{ } \} \\
 & [\![\, [[a],x] \,]\!] \quad .
\end{array}
$$

The expression $[[a],x]$ satisfies Q because of lemma 7.3.0. Thus, we obtain the following definition for ; and, similarly, for ⁏ :

$$
\begin{array}{lll}
 & a\,;[x,y] & = (\; y \neq [\,] \rightarrow [\, a\,;x \,,\, y \,] \\
 & & \;\;\;\; [\!] \;\; y = [\,] \rightarrow [\, [a] \,,\, x \,] \\
 & & \;\;\;\;) \\
\& & [y,x]\,⁏\,a & = (\; y \neq [\,] \rightarrow [\, y \,,\, a\,;x \,] \\
 & & \;\;\;\; [\!] \;\; y = [\,] \rightarrow [\, x \,,\, [a] \,] \\
 & & \;\;\;\;) \quad .
\end{array}
$$

The (normal) time complexity of these definitions is $O(1)$; in order that their amortized time complexity is $O(1)$ too, the credit function must be chosen such that its value increases by a bounded amount under these operations.

7.5 Left and right tail

We now derive definitions for ltl and rtl . These derivations do not yield efficient definitions, but they do provide information on how the credit function can be chosen such that these definitions have $O(1)$ amortized time complexity.

For ltl , we derive:

$\llbracket$ ltl$\cdot$[x,y] $\rrbracket$
= { specification of ltl }
tl$\cdot\llbracket$ [x,y] $\rrbracket$
= { definition of $\llbracket\cdot\rrbracket$ }
tl$\cdot$(x#rev$\cdot$y) .

Further manipulation of this formula requires distinction of the cases $x \neq []$ and $x = []$. For the case $x = []$ we have:

tl$\cdot$(x#rev$\cdot$y)
= { x = [] }
tl$\cdot$(rev$\cdot$y)
= { #y = 1 (note0, see below) , hence tl$\cdot$(rev$\cdot$y) = [] }
[]
= { [] is the identity of # , rev$\cdot$[] = [] }
[] # rev$\cdot$[]
= { definition of $\llbracket\cdot\rrbracket$ }
$\llbracket$ [[],[]] $\rrbracket$.

Hence, for the case $x = []$ we choose ltl$\cdot$[x,y] = [[],[]] .

note0: From $Q \wedge x = []$ it follows $\#y \leq 1$. The precondition of ltl$\cdot$[x,y] is $\llbracket$ [x,y] $\rrbracket \neq []$, which equivales $x \neq [] \vee y \neq []$. This and $x = []$ implies $y \neq []$. Hence: $\#y = 1$.
□

For the case $x \neq []$ we derive:

tl$\cdot$(x#rev$\cdot$y)
= { $x \neq []$, definitions of tl and # }
tl$\cdot$x # rev$\cdot$y
= { definition of $\llbracket\cdot\rrbracket$ }
$\llbracket$ [tl$\cdot$x , y] $\rrbracket$.

So, for the case $x \neq []$, we may choose $ltl\cdot[x,y] = [\,tl\cdot x\,,\,y\,]$, provided that this expression satisfies Q . I.e. we must prove $Q \Rightarrow Q(x,y \leftarrow tl\cdot x,y)$. Assuming Q we derive:

$Q(x,y \leftarrow tl\cdot x,y)$
$=\quad$ { definition of Q }
$(1 \leqslant \#(tl\cdot x) \vee \#y \leqslant 1) \wedge (1 \leqslant \#y \vee \#(tl\cdot x) \leqslant 1)$
$=\quad$ { definitions of tl and # }
$(2 \leqslant \#x \vee \#y \leqslant 1) \wedge (1 \leqslant \#y \vee \#x \leqslant 2)$
$=\quad$ { $Q \Rightarrow 1 \leqslant \#y \vee \#x \leqslant 2$ }
$(2 \leqslant \#x \vee \#y \leqslant 1)$
$\Leftarrow\quad$ { predicate calculus }
$2 \leqslant \#x$.

So, for the special case that x has at least 2 elements the above definition for ltl is correct. Remains the case that x is a singleton list:

$tl\cdot x \mathbin{+\!\!+} rev\cdot y$
$=\quad$ { $\#x = 1$ }
$[] \mathbin{+\!\!+} rev\cdot y$
$=\quad$ { [] is the identity of $\mathbin{+\!\!+}$ }
$rev\cdot y$
$=\quad$ { introduce u and v such that $y = u \mathbin{+\!\!+} v$ (note1, see below) }
$rev\cdot(u \mathbin{+\!\!+} v)$
$=\quad$ { list calculus }
$rev\cdot v \mathbin{+\!\!+} rev\cdot u$
$=\quad$ { definition of $[\![\cdot]\!]$ }
$[\![\,[rev\cdot v\,,\,u]\,]\!]$.

So, for the case $\#x = 1$ we may choose $ltl\cdot[x,y] = [rev\cdot v\,,\,u]$ where $u \mathbin{+\!\!+} v = y$.

note1: The decision to split y into parts u and v is inspired by the desire to transform $rev\cdot y$ into a pair of values. By not further specifying u

and v we retain the freedom to choose the most efficient representation.
□

Evaluation of $[\text{rev}\cdot v, u]$ takes $O(\#y)$ time, independently of how u and v have been chosen. In order to obtain $O(1)$ amortized time complexity, the value of the credit function must decrease by an amount that is at least linear in $\#y$. I.e. $c\cdot[x,y] - c\cdot[\text{rev}\cdot v,u]$ must be linear in $\#y$, where c denotes the credit function.

7.6 The credit function

In order not to disturb the symmetry we require c to be symmetric in x and y; i.e. $c\cdot[x,y] = c\cdot[y,x]$, for all x and y. One of the simplest such functions is given by:

$$c\cdot[x,y] \;=\; \#x + \#y \quad .$$

By a simple calculation it can be shown that this definition is equivalent to:

$$c\cdot[x,y] \;=\; \#[\![\,[x,y]\,]\!] \quad .$$

This function is not useful, for two reasons. First, the length of the represented list increases or decreases by 1 only under each of the list operations. So, amortized and normal complexity coincide. Phrased differently, the idea of amortized complexity amounts to choosing a function c that allows, every now and then, more substantial decreases of its value. Second, the second definition shows that c is invariant under changes of representation; so, this c gives no heuristic guidance when we exploit the freedom to apply changes of representation.

A function c that does satisfy these requirements is:

$$c\cdot[x,y] \;=\; |\,\#x - \#y\,| \quad .$$

We leave the proof that the value of c increases by at most 1 under the left and right cons operations as an exercise to the reader.

We now use this c to complete the design of the definitions for ltl and rtl. In the previous section we have derived that, for the special case $\#x=1$, values u and v must be chosen such that $c\cdot[x,y] - c\cdot[rev\cdot v,u]$ is linear in $\#y$. We have:

$$
\begin{array}{ll}
 & c\cdot[x,y] \\
= & \quad \{\text{ definition of } c \ \} \\
 & |\,\#x - \#y\,| \\
= & \quad \{\ \#x = 1\ \} \\
 & |\,\#y - 1\,| \\
\geqslant & \quad \{\text{ definition of } |\cdot| \ \} \\
 & \#y - 1 \qquad .
\end{array}
$$

This formula is linear in $\#y$. Hence, the decrease of c is linear in $\#y$, provided that we see to it that $c\cdot[rev\cdot v,u]$ is not too large:

$$
\begin{array}{ll}
 & c\cdot[rev\cdot v,u] \\
= & \quad \{\text{ definition of } c \ \} \\
 & |\,\#(rev\cdot v) - \#u\,| \\
= & \quad \{\text{ properties of } \# \text{ and } rev \ \} \\
 & |\,\#v - \#u\,| \qquad .
\end{array}
$$

By choosing the lengths of u and v as equal as possible we achieve that $c\cdot[rev\cdot v,u] \leqslant 1$. Therefore, we choose u and v such that:

$$u +\!\!+\, v = y \ \wedge\ \#u \leqslant \#v \leqslant \#u{+}1 \qquad .$$

The pair $[rev\cdot v,u]$ thus specified satisfies Q; this can be shown by a simple calculation. From this specification it follows that $\#u = \#y\ \mathbf{div}\ 2$; hence, we have $u = y{\uparrow}k$ and $v = y{\downarrow}k$ where $k = \#y\ \mathbf{div}\ 2$.

Putting all pieces together we obtain the following definitions for ltl and its symmetric counterpart rtl:

```
    ltl·[x,y] = ( #x = 0 → [ [] , [] ]
                [] #x = 1 → [ rev·(y↓k) , y↑k ] |[ k = #y div 2 ]|
                [] #x ≥ 2 → [ tl·x , y ]
                )
  & rtl·[y,x] = ( #x = 0 → [ [] , [] ]
                [] #x = 1 → [ y↑k , rev·(y↓k) ] |[ k = #y div 2 ]|
                [] #x ≥ 2 → [ y , tl·x ]
                )    .
```

7.7 Epilogue

A formalised notion of amortized complexity turns out to be of heuristic value for the derivation of efficient programs. In our example, we have chosen function c with no more justification than an appeal to a few general criteria. Once c has been chosen, the definitions for ltl and rtl can be completed in a rather straightforward way.

With the list representation used in this chapter, reversal of a list becomes a trivial operation: we have rev·⟦ [x,y] ⟧ = ⟦ [y,x] ⟧ , for all x and y . Hence Rev |[Rev·[x,y] = [y,x]]| is a correct and efficient implementation of rev .

8 On the computation of prime numbers

8.0 Introduction

Many authors on functional programming present functional programs for *Eratosthenes's sieve* for the computation of prime numbers, but hardly anybody supplies a formal proof of correctness, let alone a derivation. In this chapter we present a formal derivation that gives rise to two different encodings of essentially the same algorithm.

We use some of the results from chapter 6; particularly, we use the technique to represent functions on Nat by lists (section 6.4), the notations for the representation of sets by lists (section 6.5), and function flt ("filter") for the computation of subsets of infinite sets (section 6.7).

8.1 Specification

We use the name *multiples* for the natural numbers that are at least 2 . Throughout this chapter x and y denote multiples, and i and j denote naturals. *"x is not a divisor of y"* is denoted by $x \nmid y$. The *primes* are the multiples that satisfy predicate P , with:

(0a) $P{\cdot}x \equiv (\mathrm{A}\, y : y<x : y \nmid x)$.

Relation $\nmid$ is such that the set of primes is infinite. Apart from this, no properties of $\nmid$ are needed to use this definition. Because the range of the quantification in (0a) is finite, $P{\cdot}x$ is computable for each multiple x . Definition (0a) is equivalent to the following recursive definition of P :

(0b) $P{\cdot}x \equiv (\mathrm{A}\, y : y<x \wedge P{\cdot}y : y \nmid x)$.

The proof of equivalence of these two definitions requires knowledge of some properties of $\nmid$, but for the derivation of programs based on (0b) we do not need these properties. The advantage of (0b) over (0a) is that the range of its quantification is smaller; therefore, we expect that it is easier to derive efficient programs from (0b) than from (0a) .

Similarly, we might use the following, equally equivalent, definition of P to derive even more efficient programs:

(0c) $\quad P{\cdot}x \;\equiv\; (\mathbf{A}\, y : y^2 \leqslant x \wedge P{\cdot}y : y \nmid x)$.

For the sake of simplicity, however, we restrict ourselves to the use of (0b) .

We are interested in a program for the increasing infinite list p representing the set of primes; i.e. apart from being an infinite list, p has to satisfy $(1) \wedge (2)$, with:

(1) $\quad (\mathbf{A}\, i,j :: i<j \equiv p{\cdot}i<p{\cdot}j)$

(2) $\quad (\mathbf{A}\, x :: P{\cdot}x \equiv (\mathbf{E}\, i :: p{\cdot}i=x))$.

Of course, we could have specified p also by $p = L{\cdot}P$, where L is the function defined in section 6.5. Although it is possible to apply the technique developed in section 6.5, we shall not do so; instead, inspired by the recursive form of (0b) , we derive a recursive definition for p ; next, by means of the technique of section 6.4, we transform this function into an infinite list. Because there are infinitely many primes, the above specification is meaningful.

Formulae (1) and (2) have been chosen to suit the way we use them in our derivations. Which form is best-suited for this purpose can only be discovered during the process of program derivation, by observation of what is needed. This illustrates that the formalisation of a program's specification and the derivation of the program itself often have to be carried out hand-in-hand [Dij4]. Another example of this phenomenon is given in chapter 9.

8.2 Derivation

The purpose of the following derivation is to obtain a characterisation of $p{\cdot}j$ in terms of $p{\cdot}i$, for $i : i<j$ only. We start with the observation that (3) follows, by instantiation, from (2) , and that, for p satisfying (2) , (0b) is equivalent to (4) , with:

(3) $(\mathbf{A}\, i :: P.(p.i))$
(4) $P.x \equiv (\mathbf{A}\, i : p.i<x : p.i \nmid x)$.

We derive:

$$
\begin{array}{ll}
 & P.(p.j) \\
= & \quad \{ (4) \} \\
 & (\mathbf{A}\, i : p.i<p.j : p.i \nmid p.j) \\
= & \quad \{ (1) \} \\
 & (\mathbf{A}\, i : i<j : p.i \nmid p.j) \quad .
\end{array}
$$

Moreover, $p.j$ must also satisfy $(\mathbf{A}\, i : i<j : p.i<p.j)$; this follows directly from (1) . Hence, $p.j$ is a solution of the equation $x : (5a) \wedge (5b)$, with:

(5a) $(\mathbf{A}\, i : i<j : p.i<x)$
(5b) $(\mathbf{A}\, i : i<j : p.i \nmid x)$.

The question now arises *what* solution $p.j$ is; we prove that it is the smallest solution, by case analysis. For x satisfying $(5a) \wedge (5b)$, we have:

$$
\begin{array}{ll}
 & P.x \\
= & \quad \{ (2) \} \\
 & (\mathbf{E}\, i :: p.i=x) \\
= & \quad \{ \text{range split} \} \\
 & (\mathbf{E}\, i : i<j : p.i=x) \vee (\mathbf{E}\, i : j \leq i : p.i=x) \\
= & \quad \{ (5a) \Rightarrow \neg(\mathbf{E}\, i : i<j : p.i=x) \} \\
 & (\mathbf{E}\, i : j \leq i : p.i=x) \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & \neg P.x \\
= & \quad \{ (4) \text{ ; de Morgan } \} \\
 & (\mathbf{E}\, i : p.i<x : \neg(p.i \nmid x)) \\
= & \quad \{ \text{range split} \}
\end{array}
$$

$(\mathbf{E}\, i : i<j \wedge p{\cdot}i<x : \neg(p{\cdot}i \nmid x)) \vee (\mathbf{E}\, i : j\leqslant i \wedge p{\cdot}i<x : \neg(p{\cdot}i \nmid x))$

$=$ { (5b) $\Rightarrow \neg(\mathbf{E}\, i : i<j \wedge p{\cdot}i<x : \neg(p{\cdot}i \nmid x))$ }

$(\mathbf{E}\, i : j\leqslant i \wedge p{\cdot}i<x : \neg(p{\cdot}i \nmid x))$

$\Rightarrow$ { predicate calculus }

$(\mathbf{E}\, i : j\leqslant i : p{\cdot}i<x)$.

Hence, in either case we have $(\mathbf{E}\, i : j\leqslant i : p{\cdot}i\leqslant x)$, and:

$(\mathbf{E}\, i : j\leqslant i : p{\cdot}i\leqslant x)$

$=$ { trading }

$(\mathbf{E}\, i :: j\leqslant i \wedge p{\cdot}i\leqslant x)$

$=$ { (1) (using $A \equiv B \equiv \neg A \equiv \neg B$) }

$(\mathbf{E}\, i :: p{\cdot}j\leqslant p{\cdot}i \wedge p{\cdot}i\leqslant x)$

$=$ { calculus }

$p{\cdot}j \leqslant x$.

So, we have $(5a) \wedge (5b) \Rightarrow p{\cdot}j \leqslant x$; hence, we conclude (6) , with:

(6) $p{\cdot}j \;=\; (\mathbf{MIN}\, x ; Q{\cdot}j{\cdot}x : x)$

(7) $Q{\cdot}j{\cdot}x \;\equiv\; (\mathbf{A}\, i : i<j : p{\cdot}i<x) \wedge (\mathbf{A}\, i : i<j : p{\cdot}i \nmid x)$.

We conclude this section with the remark that the whole development is complicated a little by our (deliberate) intention not to use properties of $\nmid$: actually, $\nmid$ is such that $(\mathbf{A}\, i : i<j : p{\cdot}i \nmid x) \Rightarrow (\mathbf{A}\, i : i<j : p{\cdot}i<x)$.

8.3 The first program

From (7) we can derive the following recursive definition of Q :

(8a) $Q{\cdot}0{\cdot}x \;\equiv\; true$

(8b) $Q{\cdot}(j+1){\cdot}x \;\equiv\; Q{\cdot}j{\cdot}x \wedge p{\cdot}j<x \wedge p{\cdot}j \nmid x$.

Formula (8b) can be simplified a little, at the expense of some case analysis:

$$
\begin{array}{ll}
 & Q\cdot(j+1)\cdot(p\cdot j) \\
= & \quad \{\ (8b)\ ;\ \neg(p\cdot j<p\cdot j)\ \} \\
 & \text{false} \quad ,
\end{array}
$$

and, for $x : p\cdot j<x$:

$$
\begin{array}{ll}
 & Q\cdot(j+1)\cdot x \\
= & \quad \{\ (8b)\ ;\ p\cdot j<x\ \} \\
 & Q\cdot j\cdot x \wedge p\cdot j \nmid x \quad .
\end{array}
$$

Now suppose that set $Q\cdot j$ is represented, -- in the sense of section 6.5 -- by an increasing infinite list q , say. Because q is increasing we conclude, using (6) :

$$p\cdot j \ = \ q\cdot 0$$

Furthermore, we have:

$$
\begin{array}{ll}
 & Q\cdot(j+1) \\
= & \quad \{\ \text{see above, now in terms of set operations}\ \} \\
 & \{x \mid Q\cdot j\cdot x \wedge p\cdot j<x \wedge p\cdot j\nmid x\} \\
= & \quad \{\ Q\cdot j = [\![q]\!]\ \} \\
 & \{q\cdot i \mid 0\leq i \wedge p\cdot j<q\cdot i \wedge p\cdot j\nmid q\cdot i\} \\
= & \quad \{\ \text{range split}\ \} \\
 & \{q\cdot 0 \mid p\cdot j<q\cdot 0 \wedge p\cdot j\nmid q\cdot 0\} \cup \{q\cdot(i+1) \mid 0\leq i \wedge p\cdot j<q\cdot(i+1) \wedge p\cdot j\nmid q\cdot(i+1)\} \\
= & \quad \{\ p\cdot j=q\cdot 0\ ,\ \text{so}\ \neg(p\cdot j<q\cdot 0) \wedge p\cdot j<q\cdot(i+1)\ ,\ q\cdot(i+1)=(q\downarrow 1)\cdot i\ \} \\
 & \{(q\downarrow 1)\cdot i \mid 0\leq i \wedge q\cdot 0\nmid(q\downarrow 1)\cdot i\} \\
= & \quad \{\ \text{specification of flt , see below}\ \} \\
 & [\![\ \text{flt}\cdot(q\downarrow 1)\ ]\!] \quad .
\end{array}
$$

Here, we have used function flt ("filter"), as discussed in section 6.7, with $(q\cdot 0\nmid)$ for the boolean function defining the subset of its parameter. flt has type $L_\infty \to L_\infty$; we recall its specification here. In it, $q\cdot 0$ occurs as a global constant:

$$\text{inc}\cdot s \Rightarrow \text{inc}\cdot(\text{flt}\cdot s) \wedge [\![\, \text{flt}\cdot s \,]\!] = \{\, s\cdot i \mid 0 \leq i \wedge q\cdot 0 \nmid s\cdot i \,\} \quad .$$

So, if q represents Q·j then flt·(q↓1) represents Q·(j+1) . More-over, from (8a) it follows that Q·0 is represented by (the increasing infinite list) from·2 |[from·i = i ; from·(i+1)]| .

We now introduce function sieve with specification:

$$(\text{A}\, q,j : L_{\infty}\cdot q : \text{inc}\cdot q \wedge [\![q]\!] = Q\cdot j \;\Rightarrow\; \text{sieve}\cdot q = p{\downarrow}j\,) \quad .$$

We have:

```
    sieve·q
=       { specification of sieve, assuming inc·q ∧ [[q]] = Q·j }
    p↓j
=       { ↑↓-calculus }
    p·j ; p↓(j+1)
=       { see above , specification of sieve }
    q·0 ; sieve·(flt·(q↓1))     .
```

Thus, we obtain the following program, well-known as Eratosthenes's sieve:

```
program0:   sieve·(from·2)
            |[ sieve·(x ; q)  =  x ; sieve·(flt·q)
                                 |[ flt·(y ; s) = (   x∤y  → y ; flt·s
                                                  [] ¬(x∤y) →     flt·s
                                                  )
                                 ]|
            & from·i          =  i ; from·(i+1)
            ]|
```
□

For our particular relation ∤ , expression (x∤y) can be rewritten as $y\,\textbf{mod}\,x \neq 0$. Actually, it is possible to avoid the use of **div** and **mod** by construction of a special-purpose version of flt·s : flt·s is the list obtained from s by omission of all *multiples* of x .

8.4 The second program

The derivation of our second program is based on the observation that predicate $Q{\cdot}j$ consists of two conjuncts that can be treated differently. For this purpose, we name these two conjuncts:

(9a) $Qa{\cdot}j{\cdot}x \;\equiv\; (\mathbf{A}\, i : i<j : p{\cdot}i<x\,)$

(9b) $Qb{\cdot}j{\cdot}x \;\equiv\; (\mathbf{A}\, i : i<j : p{\cdot}i \nmid x\,)$.

Then, we have $p{\cdot}j = (\mathbf{MIN}\, x : Qa{\cdot}j{\cdot}x \wedge Qb{\cdot}j{\cdot}x : x\,)$. Moreover, Qa and Qb have the following properties:

$Qa{\cdot}j{\cdot}x \Rightarrow Qa{\cdot}j{\cdot}(x+1)$

$(\mathbf{MIN}\, x : Qa{\cdot}0{\cdot}x : x\,) = 2$

$(\mathbf{MIN}\, x : Qa{\cdot}(j+1){\cdot}x : x\,) = p{\cdot}j + 1$

$Qb{\cdot}j{\cdot}x$ is computable, provided that $p{\uparrow}j$ has been computed .

From these observations we conclude that $p{\cdot}j$ can be computed by means of Linear Search; i.e. we introduce function g with the following specification:

$g{\cdot}x \;=\; (\mathbf{MIN}\, y : x \leq y \wedge Qb{\cdot}j{\cdot}y : y\,)$.

Then, we have $p{\cdot}j = g{\cdot}(\mathbf{MIN}\, x : Qa{\cdot}j{\cdot}x : x)$. In order to transform p into a list, we introduce function f with specification:

$(\mathbf{A}\, j :: f{\cdot}(\mathbf{MIN}\, x : Qa{\cdot}j{\cdot}x : x){\cdot}j = p{\downarrow}j\,)$.

Here, we have equipped f with an additional parameter right away: somehow, we need a simple representation of $(\mathbf{MIN}\, x : Qa{\cdot}j{\cdot}x : x)$. Using the above observations and the technique of section 6.4, we obtain the following program:

```
program1:    p |[ p = f·2·0 |[ f·y·j = gy ; f·(gy+1)·(j+1)
                                    |[ gy  = g·y
                                    & g·x = (  Qb·j·x → x
                                             [] ¬Qb·j·x → g·(x+1)
                                             )
                                    ]|
                            ]|
             ]|
```

□

The introduction of name p in this program is necessary because of the recursive occurrences of p in expression Qb·j·x . Qb·j·x can be eliminated from this program, in various ways, by means of the elementary programming techniques. We leave this as an exercise to the reader. One of the possible programs that can be obtained in this way is:

```
program2:    p |[ p = f·2·0
                   |[ f·y·j = gy ; f·(gy+1)·(j+1)
                           |[ gy        = g1·p·0·y
                           & g1·q·k·x = ( k = j → x
                                         [] k < j →
                                            (   q·0∤x → g1·(q↓1)·(k+1)·x
                                            [] ¬(q·0∤x) → g1·p·0·(x+1)
                                            )
                                         )
                           ]|
                   ]|
             ]|
```

□

8.5 Epilogue

The derivations of program0 and program2 have a large part in common. Yet, the two programs look very different. Both programs can, however, be evaluated in such a way that the *same* ∤ operations are performed in the *same* order. In this respect, both programs can be considered as realisations of the same abstract algorithm, the essence of which is captured by formulae (6) and (7) . On the other hand, the two programs exhibit some important differences.

Program0 is simpler and shorter than program2. The former also is more abstract than the latter. In program0, for instance, infinite lists are more heavily used than in program2; moreover, its evaluation depends more on the use of lazy evaluation than program2 does. We illustrate this as follows. Suppose that we are interested in p↑n , i.e. the finite list containing the first n primes, only. In order to account for this, we modify program2 by replacing the definition of function f by:

```
f·y·j  =  ( j = n → []
          ▯ j < n → gy  ;  f·(gy+1)·(j+1)
                    |[ ... ]|
          )    .
```

Now, the program contains no infinite lists and the only laziness required during program evaluation is that no guarded expression is evaluated before evaluation of its guard has yielded value true . In program0, p↑n equals sieve·(from·2)↑n ; because this value is computable, there exists a natural m such that sieve·(from·2)↑n = sieve·(from·2↑m)↑n ; thus, the infinite lists can be eliminated from program0. It suffices to extend the definitions of sieve and flt with sieve·[] = [] and flt·[] = [] . The problem is, however, that it requires quite some number-theoretic knowledge to determine a sufficiently small estimate for m . As a result, it is easier to obtain a sequential program from program2 than from program0; because the definitions of f and g1 are tail recursive, program2 almost *is* a sequential program.

From the above we conclude that, when the program is intended to be executed by a functional-program evaluator, program0 is to be preferred; when, however, the functional program is only used as a stepping-stone in the development of a sequential program, program0 is better avoided. For the time being, we do not know whether program2 can be derived, by simple program transformations, from program0.

The programs derived in this chapter are correct for any relation $\nmid$ such that set P , as defined by (0b) , is infinite. For example, by replacing $\nmid$ by $|$ ("divides") we obtain the set of all proper powers of 2 .

9 Minimal enumerations

9.0 Introduction

The problem to be solved is the design of a program for the computation of a, so-called, *minimal enumeration* of (the elements of) a given *bag*. Such a program can be used to solve problems as the well-known *travelling-salesman problem*. The jargon for this kind of programs comprises terms like *backtracking* and *branch-and-bound techniques*. Here we intend to show, by example, that such programs can be derived from their specifications by, more or less straightforward, calculation.

We present derivations of two functional programs; the first program can be characterised as straightforward backtracking, whereas the second program, a refinement of the first one, contains an application of the branch-and-bound idea. Finally, we show that these programs can be implemented quite easily as sequential programs.

9.1 Specification

Throughout the discussion of this example, we adopt the following convention for the types of the variables used:

x,y	: finite bag (of elements) ,
s,t,u	: finite list (of elements) ,
a,b	: element (of either bags or lists) ,
d,e	: Int .

The type of the elements of the bags and lists is irrelevant for our discussion. For bags, we use the following notational conventions. The *empty bag* is denoted by $\emptyset$. The *singleton bag* containing element a (once) is denoted by $\{a\}$. *Bag summation* and *subtraction* are denoted by $+$ and $-$.

Lists can be considered as representations of bags -- by enumeration of the elements -- . The abstraction function, mapping lists to the bags they represent, is denoted by $[\![\cdot]\!]$; i.e, informally, we have:

$$[\![s]\!] = \text{"the bag represented by s"} \quad .$$

Gradually, i.e. not until the need arises, we replace this definition by a more formal one. We call list s an *enumeration* of bag x if $[\![s]\!] = x$.

The problem can now be stated as follows. For C a fixed function, of type $L_* \rightarrow \mathrm{Int}$, we are interested in function f specified by:

(0) $\quad f{\cdot}x = (\mathbf{MIN}\, s : [\![s]\!] = x : C{\cdot}s) \quad .$

According to (0) , we confine our attention to the minimal value of C over all enumerations of x . The programs thus obtained can be easily modified to include the computation of a specimen of such a minimal enumeration; we leave this as an exercise to the interested reader.

9.2 The first program

We use induction on the size of x :

$$\begin{array}{ll} & f{\cdot}\emptyset \\ = & \quad \{ (0) \} \\ & (\mathbf{MIN}\, s : [\![s]\!] = \emptyset : C{\cdot}s) \\ = & \quad \{ (1),\ \text{see below} \} \\ & (\mathbf{MIN}\, s : s = [\,] : C{\cdot}s) \\ = & \quad \{ \text{one-point rule} \} \\ & C{\cdot}[\,] \quad . \end{array}$$

The second step in this derivation is correct, provided that $[\![\cdot]\!]$ satisfies:

(1) $\quad [\![s]\!] = \emptyset \;\equiv\; s = [\,] \quad .$

Formula (1) is part of the (formal) definition of $[\![\cdot]\!]$. For x , $x \neq \emptyset$, we derive:

$$\begin{array}{ll}
 & f\cdot x \\
= & \quad \{\ (0)\ \} \\
 & (\mathbf{MIN}\, s : [\![s]\!] = x : C\cdot s) \\
= & \quad \{\ [\![s]\!]=x \wedge x\neq\emptyset \Rightarrow \{(1)\}\ s\neq[\,]\ ,\ \text{definition of nonempty lists}\ \} \\
 & (\mathbf{MIN}\, a,t : [\![a\,;t]\!] = x : C\cdot(a\,;t)) \\
= & \quad \{\ (2),\ \text{see below}\ \} \\
 & (\mathbf{MIN}\, a,t : a\in x \wedge [\![t]\!] = x-\{a\} : C\cdot(a\,;t)) \\
= & \quad \{\ \text{nesting dummies}\ \} \\
 & (\mathbf{MIN}\, a : a\in x : (\mathbf{MIN}\, t : [\![t]\!] = x-\{a\} : C\cdot(a\,;t))\ ) \quad .
\end{array}$$

Formula (2) is the second part of the definition of $[\![\cdot]\!]$, as we need it:

$$(2) \qquad [\![a\,;t]\!] = x \;\equiv\; a\in x \wedge [\![t]\!] = x-\{a\} \quad .$$

That we are heading for a definition of $[\![\cdot]\!]$ in the form of a few equivalences is not surprising. Such a definition enables us to replace the range of the quantified expression by another one without affecting the value of the expression. From (0) we see that we are *only* interested in a definition of the solutions of the equation $s\colon [\![s]\!] = x$, for given x . (1) and (2) provide such a definition, in a recursive way.

The term $(\mathbf{MIN}\, t : [\![t]\!] = x-\{a\} : C\cdot(a\,;t))$ of the expression derived above resembles the expression in (0) , but it is not an instance of it; thus, recursive application of (0) is prohibited. The difference lies in the subexpressions s and $a\,;t$, as they occur as arguments of C . Both are, however, instances of an expression of the form $u +\!\!+ s$. By generalisation by abstraction we obtain the following specification for function $f1$, a generalisation of f :

$$(3) \qquad f1\cdot u\cdot x = (\mathbf{MIN}\, s : [\![s]\!] = x : C\cdot(u +\!\!+ s)) \quad .$$

Hence, we may use $f1\cdot[\,]$ for f . The above derivation can now be redone as follows:

$$\begin{array}{ll}
 & f1\cdot u\cdot\emptyset \\
= & \quad \{\ \text{as before (mutatis mutandis)}\ \} \\
 & C\cdot u \quad .
\end{array}$$

For x , $x \neq \emptyset$;

```
     f1·u·x
  =       { as before (mutatis mutandis) }
     (MIN a : a∈x : (MIN t : ⟦t⟧ = x−{a} : C·(u++(a ; t)) ) )
  =       { a ; t = [a]++t , ++ is associative }
     (MIN a : a∈x : (MIN t : ⟦t⟧ = x−{a} : C·((u++[a])++t) ) )
  =       { induction hypothesis (3) }
     (MIN a : a∈x : f1·(u++[a])·(x−{a}) )
  =       { introduction of function  g , with specification (4) }
     g·x   .
```

(4) $\quad$ g·y = (MIN a : a∈y : f1·(u++[a])·(x−{a})) , for y: y⊆x $\quad$.

The specification of g is obtained by replacement of constant x by a variable. Notice that we have replaced the first occurrence of x only. From (4) , a recursive definition for g can be obtained by straightforward calculation. Thus, we obtain our first program. Operationally, this program can be considered as a backtracking algorithm: in parameter u all enumerations of the bag are "built up", and evaluation of g·x generates recursive applications f1·(u++[b])·(x−{b}) , for all b in x .

program0:

```
   f1·[]
   |[ f1·u·x = ( x=∅ → C·u
               [] x≠∅ → g·x |[ g·y = ( y=∅ → ∞
                                     [] y≠∅ → f1·(u++[b])·(x−{b}) min g·(y−{b})
                                              |[ b : b∈y ]|
                                     )
                          ]|
               )
   ]|
```

□

remark 9.2.0: The where-clause $\|[\ b : b\in y\ ]\|$ does not specify *which* element of y is to be chosen. In this stage of the design the choice is irrelevant. When it comes to the implementation of the program, and, particularly, to the choice of a representation of bags, this freedom can be exploited.

Value ∞ in this program is used as the identity of **min** . It may be replaced by any integer d satisfying: $(\mathbf{A}\, s : [\![s]\!] = x : C{\cdot}s < d)$; d may depend on x . Alternatively, at the expense of slightly longer code, ∞ can be eliminated from the program by means of a simple transformation.

□

9.3 The second program

Program0 contains several possibilities for further manipulation. The definition of g , for instance, is linearly recursive and it contains an associative operator, **min** , with identity ∞ . Hence, we may apply the Tail Recursion Theorem and replace the subexpression $g{\cdot}x\ \|[\ g{\cdot}y = \ldots\ ]\|$ by the following, equivalent, expression:

```
(5)     g1·∞·x |[ g1·d·y = ( y = ∅  →  d
                           [] y ≠ ∅  →  g1·(d min f1·(u++[b])·(x−{b}))·(y−{b})
                                          |[ b : b∈y ]|
                           )
                 ]|   .
```

Notice that $g1$ can be specified in terms of g as follows:

(6) $\quad g1{\cdot}d{\cdot}y = d\ \mathbf{min}\ g{\cdot}y$, for y: $y \subseteq x$.

remark 9.3.0: The above step and the next one can be performed in either order. The order of these steps is irrelevant to the extent that the same final solution can be obtained in both ways. We have chosen the order that gives rise to the shortest presentation, but the only way to know this seems to be to try both. Generally, larger examples involve longer formulae, and long formulae offer greater manipulative freedom than short ones; the problem how to choose the next step grows accordingly.

□

The following transformation serves to increase the efficiency of the program. In this respect, subexpression $d\ \mathbf{min}\ f1\cdot(u+\!\!+[b])\cdot(x-\{b\})$ is of interest: its value is d whenever $d \leqslant f1\cdot(u+\!\!+[b])\cdot(x-\{b\})$; a potentially more efficient program is obtained if we can find a way to establish the truth of this inequality without evaluation of its right-hand side. Therefore, we investigate the following generalisation of $f1$, called $f2$ -- so that $f1\cdot u\cdot x = f2\cdot\infty\cdot u\cdot x$ -- :

(7) $\quad f2\cdot e\cdot u\cdot x = e\ \mathbf{min}\ f1\cdot u\cdot x \quad .$

We then have:

$$e \leqslant f1\cdot u\cdot x \quad\Rightarrow\quad f2\cdot e\cdot u\cdot x = e \quad .$$

We derive:

$\qquad e \leqslant f1\cdot u\cdot x$

$=\qquad \{\ (3)\ ;\ \text{property of}\ \mathbf{MIN}\ \}$

$\qquad (\mathbf{A}\, s : [\![s]\!] = x\ :\ e \leqslant C\cdot(u+\!\!+s)\ )$

$\Leftarrow\qquad \{\ \text{assume (8), see below}\ \}$

$\qquad (\mathbf{A}\, s : [\![s]\!] = x\ :\ e \leqslant D\cdot u\ )$

$=\qquad \{\ \text{predicate calculus}\ \}$

$\qquad e \leqslant D\cdot u \quad .$

The purpose of this derivation is to obtain an *approximation* of $C\cdot(u+\!\!+s)$ that does not depend on s . We therefore assume the availability of a function D , of type $L_{*} \to \text{Int}$, such that $D\cdot u$ provides a lower bound for $C\cdot(u+\!\!+s)$, for all s ; i.e. D is assumed to satisfy:

(8) $\quad (\mathbf{A}\, s : [\![s]\!] = x\ :\ D\cdot u \leqslant C\cdot(u+\!\!+s)\ ) \quad .$

We conclude:

(9) $\quad e \leqslant D\cdot u \quad\Rightarrow\quad f2\cdot e\cdot u\cdot x = e \quad .$

Furthermore, we derive:

$\qquad f2\cdot e\cdot u\cdot \emptyset$

$=\qquad \{\ (7)\ \}$

```
      e min f1·u·∅
   =      { unfolding f1 (program0) }
      e min C·u   ,
```

and for x , $x \neq \emptyset$:

```
      f2·e·u·x
   =      { (7) }
      e min f1·u·x
   =      { unfolding f1 (program0, x ≠ ∅) }
      e min g·x
   =      { (6) }
      g1·e·x  .
```

We replace in (5) expression d **min** f1·(u++[b])·(x−{b}) by the equivalent f2·d·(u++[b])·(x−{b}) (using (7)). Since the values of d emerging during the computation of g1·e·x form a descending sequence, the chance that, during evaluation of f2·d·(u++[b])·(x−{b}) , (9) can be applied is likely to increase as the computation of g1·e·x proceeds.

Putting all pieces together and eliminating name f1 we obtain our second program.

program1:

```
   f2·∞·[]
   |[ f2·e·u·x = ( e ≤ D·u          → e
                 [] e > D·u ∧ x = ∅ → e min C·u
                 [] e > D·u ∧ x ≠ ∅ →
                     g1·e·x |[ g1·d·y = ( y = ∅ → d
                                        [] y ≠ ∅ → g1·(f2·d·(u++[b])·(x−{b}))·(y−{b})
                                                     |[ b : b∈y ]|
                                        )
                            ]|
                 )
   ]|
```
□

This program can be considered as a branch-and-bound algorithm: by means of the alternative $e \leqslant D{\cdot}u \rightarrow e$ the, so-called, *search space* of the algorithm is bounded. Hence, this program may be expected to be more efficient than program0; in the worst case, however, the two programs are equivalent.

9.4 Implementation

Using program1 as a starting point, we now construct a sequential program for f, i.e. for $f2{\cdot}\infty{\cdot}[\,]$. This is easy, provided that we permit ourselves the use of a recursive function for the implementation of $f2$: the pattern of recursion of $f2$ is too complicated to be simply translatable into iterative form. The sequential versions of f and $f2$ are called F and $F2$ respectively. The (tail recursive) definition of $g1$, on the other hand, can straightforwardly be recoded as an iteration.

To make the implementation a little more interesting, we also choose a representation of variables u, x, and y: instead of lists and bags we use arrays. The choice of a suitable representation can be made in many ways. Because the design of such representations is not the subject of this study, we simply present one of the possibilities, with no other heuristic justification than that it is simple and reasonably efficient. We proceed in a number of steps:

- Observing that $[\![u +\!\!+ [b]]\!] + (x - \{b\}) = [\![u]\!] + x$, we conclude that, during evaluation of $f2{\cdot}e{\cdot}u{\cdot}x$, the value of $[\![u]\!] + x$ is constant, and, hence, equal to the initial value of x (because, initially, $u = [\,]$). So, we decide to represent u and x together by an array $s{\cdot}i(0 \leqslant i < N)$ and a natural n, according to the following invariant:

$$0 \leqslant n \leqslant N \;\wedge\; u = s{\cdot}i(0 \leqslant i < n) \;\wedge\; x = [\![s{\cdot}i(n \leqslant i < N)]\!] \quad .$$

- This representation is such that $[\![s{\cdot}i(n \leqslant i < N)]\!]$ is invariant under permutations of $s{\cdot}i(n \leqslant i < N)$. We exploit this freedom.
- Similarly, using $y \subseteq x$, we represent y by a natural m, as follows:

$$n \leqslant m \leqslant N \;\wedge\; y = [\![s{\cdot}i(m \leqslant i < N)]\!] \quad .$$

- Observing that the difference between the value to be supplied for s , in the recursive application of F2 , and the parameter s itself is rather small, we decide that parameter s can be turned into a global variable of F2 . Evaluations of F2 do not affect the value of s , although s is modified *temporarily* during these evaluations.
- For the implementations of functions C and D we have used the same names. Their parameterlists have been adapted to the changed representation of list u .

(end •)

remark 9.4.0: Some of these transformations could also have been applied to the functional program. The conversion of a parameter into a global variable, thus introducing side effects, is, by its very nature, impossible in functional programs. Not surprisingly, it seems wise to consider such transformations as optimisations to be applied only when the code has been almost completed.

□

program2:

```
    function F(s·i(0 ⩽ i < N) : array of element) : Int
    = |[  function C(n : Nat) : Int  = |[ { value:  C·u  } ]|
        ; function D(n : Nat) : Int  = |[ { value:  D·u  } ]|
        ; function F2(e : Int ; n : Nat) : Int
          = |[ if e⩽D(n)            → F2 := e
               [] e>D(n) ∧ n=N      → F2 := e min C(n)
               [] e>D(n) ∧ n<N      → |[ var d : Int ; m : Nat
                                        ; d,m := e,n
                                        { invariant: y = [[s·i(m ⩽ i < N)]] ∧ n ⩽ m ⩽ N
                                                     ∧ g1·e·x = g1·d·y  }
                                        ; do m ≠ N → { b = s·m ∧ s(n)∈x ∧ s·m∈y ∧
                                                       y−{b} = [[s·i(m+1 ⩽ i < N)]]  }
                                                       s:swap(n,m)
                                                     { u++[b] = s·i(0 ⩽ i < n+1) ∧
                                                       x−{b}  = [[s·i(n+1 ⩽ i < N)]]  }
                                                     ;  d  := F2(d,n+1)
                                                     ;  s:swap(n,m)
                                                     { y−{b} = [[s·i(m+1 ⩽ i < N)]]  }
                                                     ;  m := m + 1
                                          od { y = ∅ , hence: g1·e·x = d }
                                        ; F2 := d
                                       ]|
               fi
             ]| { end F2 }
        ; F := F2(∞,0)
      ]| { end F }
□
```

9.5 Epilogue

The only techniques used for the derivation of program0 and program1 are recursion and generalisation by abstraction, plus a little bit of common sense: although the programs and their derivations are not completely trivial -- several variations are possible -- , they are not very difficult either.

In this respect, it is questionable whether notions like "backtracking" and "branch-and-bound" are so elusive that they deserve special names.

A common way to design sequential programs for this kind of problems is to impose a total order on the set of all potential solutions, the so-called *search space*, of the problem. This order is chosen in such a way that the elements of this set can be generated, in a simple way, by means of an iterative sequential program. For example, for the problem discussed in this chapter one could use the *lexicographic order* on the set of all enumerations of the given bag. The advantage of a recursive characterisation of the search space is that the order in which its elements are computed can be left implicit. Moreover, when this order is chosen in too early a stage of the design, this may give rise to an unnecessary complicated program. For example, program2 is simpler than the corresponding iterative program based on the lexicographic order.

Functional programs contain less (explicit) information on the order in which the steps of the computation have to be performed than their sequential counterparts. In this respect, functional programs are more abstract than sequential programs. The example in this chapter shows that functional programming can be used for the design of abstract versions of an algorithm that eventually will be encoded as a sequential program. The use of functional programming might thus embody a meaningful separation of concerns: during the design of the functional program, attention is focused on the relations between the values to be computed, whereas the concern for the order in which these values will be computed is postponed until the moment the functional program is transformed into a sequential one. On the other hand, the difference between the two kind of programs is not so large -- particularly not when the sequential-program notation allows recursion -- that the gap to be bridged by the transformation is too large to be manageable.

10 Pattern matching and related problems

10.0 Introduction

In this chapter we derive a program that can be considered as the core algorithm of a number of efficient programs for *Knuth–Morris–Pratt-like* pattern matching [Knu], periodicity computation, and carré and overlap recognition. As a by-product of our derivation the, so-called, *preprocessing phase* of the KMP algorithm emerges in this design as an instance of the same problem. Despite the use of -- in comparison to array operations -- not so efficient list operations, the program has linear time complexity.

We proceed as follows. First, we derive a program without taking into account the inefficiencies of the list operations. Second, using the techniques of chapter 6, we transform this program into a more efficient one by elimination of all inefficient list operations. Finally, we show a number of applications of the program.

10.1 Specification

Without giving further justification here, we stipulate that we are interested in function mpp , of type $\mathcal{L}_\infty \to \mathcal{L}_\infty \to \mathcal{L}_\infty(\mathrm{Nat})$, with the following informal specification:

$$\mathrm{mpp}\cdot x\cdot y\cdot j \;=\; \text{"the maximal length of any prefix of } x \text{ that is a suffix of } y{\uparrow}j\text{"} \quad , \quad 0 \leqslant j \quad .$$

The elements of infinite lists x and y , in this specification, may have any type on which = is a permitted operation: test for equality will be the only operation applied to these elements.

The above specification can be formalised as follows. For finite list s , the suffixes of s are the lists $s{\downarrow}i$, for $i : 0 \leqslant i \leqslant \#s$. The suffixes of $y{\uparrow}j$ are $y{\uparrow}j{\downarrow}i$, $0 \leqslant i \leqslant j$, which have length $j-i$. By means of the dummy substitution $i \leftarrow j-i$ we obtain for the suffix of $y{\uparrow}j$ of length i :

$$y{\uparrow}j{\downarrow}(j-i)$$
$$= \quad \{ \uparrow\downarrow\text{-calculus: } y{\uparrow}(a+b){\downarrow}b = y{\downarrow}b{\uparrow}a \text{ , } j = i+(j-i) \}$$
$$y{\downarrow}(j-i){\uparrow}i \quad .$$

Thus, we obtain the following formal specification for mpp :

$$mpp{\cdot}x{\cdot}y{\cdot}j \;=\; (\mathbf{MAX}\, i : 0 \leqslant i \leqslant j \wedge x{\uparrow}i = y{\downarrow}(j-i){\uparrow}i : i) \quad , \quad 0 \leqslant j \quad .$$

10.2 The first program

In order to separate our various concerns we introduce some nomenclature. First, we expect that, throughout the discussion, mpp's parameters will occur mainly as global constants. Therefore, we abbreviate $mpp{\cdot}x{\cdot}y$ to z and we carry out the derivation in terms of z . Second, we introduce predicate C , as follows:

(C0) $\quad C{\cdot}i{\cdot}j \;\equiv\; x{\uparrow}i = y{\downarrow}(j-i){\uparrow}i \quad , \quad 0 \leqslant i \leqslant j \quad .$

Using C and mpp's specification we obtain the following specification for z . In it, the requirement that z is a list has been left implicit.

(z0) $\quad z{\cdot}j \;=\; (\mathbf{MAX}\, i : 0 \leqslant i \leqslant j \wedge C{\cdot}i{\cdot}j : i) \quad , \quad 0 \leqslant j \quad .$

This specification is meaningful, because of the following property of C :

(C1) $\quad C{\cdot}0{\cdot}j \quad , \quad 0 \leqslant j \quad .$

The expression in (z0) can be easily turned into a program by an application of Linear Search: replacement, in this expression, of the left-most occurrence of j by a parameter yields function g with specification:

$$g{\cdot}k \;=\; (\mathbf{MAX}\, i : 0 \leqslant i \leqslant k \wedge C{\cdot}i{\cdot}j : i) \quad , \quad 0 \leqslant k \leqslant j \quad .$$

In this formula, j occurs as a global constant: for this j we have $z{\cdot}j = g{\cdot}j$.

The derivation of a definition for g is straightforward. Ignoring, for the time being, that z should be a list, we thus obtain program0.

program0:

```
        z |[ z·j = g·j |[ g·k = (  C·k·j → k
                                 []¬C·k·j → g·(k−1)
                                 )
                      ]|
          ]|
```
□

Although simple and correct, this program is not efficient, for two reasons: the rather large range of the Linear Search -- evaluation of $g{\cdot}j$ requires $1+j-g{\cdot}j$ unfoldings of g -- , and the fact that evaluation of $C{\cdot}k{\cdot}j$ may require k comparisons of list elements.

In order to obtain a more efficient program, we take into account some of C's properties, in the form of the following recurrence relation:

(C2) $\quad C{\cdot}(i+1){\cdot}(j+1) \equiv C{\cdot}i{\cdot}j \wedge B{\cdot}i{\cdot}j \quad , \quad 0 \leqslant i \leqslant j$

(B) $\quad B{\cdot}i{\cdot}j \equiv x{\cdot}i = y{\cdot}j \quad , \quad 0 \leqslant i \leqslant j \quad .$

This relation cannot be exploited for $z{\cdot}0$ (see (z0)). Therefore, we need some case analysis. Using (z0) and (C1) we derive:

$$z{\cdot}0 = 0 \quad .$$

Furthermore, we derive:

$z{\cdot}(j+1)$

$=\quad$ { (z0) }

$(\mathbf{MAX}\, i : 0 \leqslant i \leqslant j+1 \wedge C{\cdot}i{\cdot}(j+1) : i)$

$=\quad$ { introduction of function g (see below) }

$g{\cdot}j \quad .$

As before, function g is introduced by replacement of the left-most occurrence of j by a parameter; its specification is:

(g) $g{\cdot}k \;=\; (\mathbf{MAX}\, i : 0 \leqslant i \leqslant k{+}1 \wedge C{\cdot}i{\cdot}(j{+}1) : i) \quad , \quad 0 \leqslant k \leqslant j$.

From this specification we obtain, again using Linear Search, program1.

program1:

```
z |[ z·0      = 0
   & z·(j+1) = g·j |[ g·k = (  C·(k+1)·(j+1)          → k + 1
                             []¬C·(k+1)·(j+1) ∧ 0=k   → 0
                             []¬C·(k+1)·(j+1) ∧ 0<k   → g·(k−1)
                             )
                   ]|
   ]|
```

□

Since g·k has precondition $0 \leqslant k$, the guard of the recursive application $g{\cdot}(k-1)$ must be strengthened with $0 < k$ and the case $0 = k$ must be dealt with separately. Hence the 3-way case analysis in the above program.

We can now apply (C2) and replace in program1 all occurrences of $C{\cdot}(k{+}1){\cdot}(j{+}1)$ by $C{\cdot}k{\cdot}j \wedge B{\cdot}k{\cdot}j$. This, of course, does not change the program's efficiency, but we gain something if we succeed in eliminating the terms $C{\cdot}k{\cdot}j$: evaluation of $B{\cdot}k{\cdot}j$ requires only 1 comparison.

About the simplest way to eliminate $C{\cdot}k{\cdot}j$ is to strenghten g's precondition with it -- or with its negation; in our case this is not sensible -- , and so we do. This is a rather bold decision: it remains to be seen whether we can get away with it.

Strengthening a function's precondition generates an additional proof obligation for each application of the function. In our case we have the applications g·j and g·(k−1) .

- g·j : We cannot guarantee $C{\cdot}j{\cdot}j$ to hold. We may, however, replace g·j by g·h , say, provided that h satisfies $(h0) \wedge (h1) \wedge (h2)$, with:

(h0) $0 \leqslant h \leqslant j$

(h1) $C{\cdot}h{\cdot}j$

(h2) $g{\cdot}j = g{\cdot}h$.

Requirement (h2) can be met as follows:

$$
\begin{array}{ll}
 & g{\cdot}j = g{\cdot}h \\
\Leftarrow & \quad \{ \text{ (g) , property } (\star) \text{ (see below) } \} \\
 & (\mathbf{A}\, i : h{+}1 < i \leqslant j{+}1 : \neg C{\cdot}i{\cdot}(j{+}1)\,) \\
= & \quad \{ \text{ dummy substitution } i \leftarrow i{+}1 \ \} \\
 & (\mathbf{A}\, i : h < i \leqslant j : \neg C{\cdot}(i{+}1){\cdot}(j{+}1)\,) \\
\Leftarrow & \quad \{ \text{ (C2) ; predicate calculus } \} \\
 & (\mathbf{A}\, i : h < i \leqslant j : \neg C{\cdot}i{\cdot}j\,) \quad .
\end{array}
$$

So, (h2) follows from the stronger (h3) , with:

(h3) $(\mathbf{A}\, i : h < i \leqslant j : \neg C{\cdot}i{\cdot}j\,)$.

Property $(\star)$ referred to in the above derivation is:

$(\star)$ For predicate P and for natural $h,j : 0 \leqslant h \leqslant j$:
$(\mathbf{A}\, i : h{<}i{\leqslant}j : \neg P{\cdot}i) \Rightarrow (\mathbf{A}\, i :: 0{\leqslant}i{\leqslant}j \wedge P{\cdot}i \equiv 0{\leqslant}i{\leqslant}h \wedge P{\cdot}i\ )$.

We have:

$$
\begin{array}{ll}
 & (h0) \wedge (h1) \wedge (h3) \\
= & \quad \{ \text{ definition of } \mathbf{MAX} \ \} \\
 & h = (\mathbf{MAX}\, i : 0 \leqslant i \leqslant j \ \wedge \ C{\cdot}i{\cdot}j : \ i\,) \\
= & \quad \{ \text{ (z0) (induction hypothesis) } \} \\
 & h = z{\cdot}j \quad .
\end{array}
$$

So, we replace $g{\cdot}j$ by $g{\cdot}(z{\cdot}j)$.

- $g{\cdot}(k{-}1)$: Here, the additional proof obligation is $C{\cdot}k{\cdot}j \Rightarrow C{\cdot}(k{-}1){\cdot}j$. We follow the same pattern of reasoning as we did for $g{\cdot}j$: we try to replace $g{\cdot}(k{-}1)$ by $g{\cdot}h$ without affecting the value of the expression and such that $0 \leqslant h \leqslant k{-}1 \wedge C{\cdot}h{\cdot}j$. By a (very) similar calculation as above we find that

(h4) $(\mathbf{MAX}\, i : 0 \leqslant i \leqslant k-1 \wedge C{\cdot}i{\cdot}j : i)$

is a suitable value for h .

Formula (h4) is a generalisation of the expression in (z0) . For arbitrary C , (h4) cannot be expressed easily in terms of z . For C defined by (C0) , however, we can perform the following calculation. This calculation constitutes the crucial invention behind the KMP-algorithm. The idea is that we try to replace j in $C{\cdot}i{\cdot}j$ by $k-1$, as a result of which an instance of (z0) is obtained. Notice that we are dealing with a recursive application of g in the definition of $g{\cdot}k$; hence, we may use precondition $C{\cdot}k{\cdot}j$. For i,k,j , $0 \leqslant i \leqslant k-1 \leqslant j-1$, we derive:

$$
\begin{array}{ll}
 & C{\cdot}i{\cdot}j \\
= & \quad \{\ (C0)\ \} \\
 & x{\uparrow}i = y{\downarrow}(j-i){\uparrow}i \\
= & \quad \{\ \text{using } C{\cdot}k{\cdot}j \text{ , see below}\ \} \\
 & x{\uparrow}i = x{\downarrow}1{\downarrow}(k-1-i){\uparrow}i \\
= & \quad \{\ (C0) \text{ with } y \leftarrow x{\downarrow}1 \text{ , let } C^* = C(y \leftarrow x{\downarrow}1)\ \} \\
 & C^*{\cdot}i{\cdot}(k-1) \qquad ,
\end{array}
$$

where:

$$
\begin{array}{ll}
 & y{\downarrow}(j-i){\uparrow}i \\
= & \quad \{\ j-i = j-k+k-i \text{ , } y{\downarrow}(a+b) = y{\downarrow}a{\downarrow}b\ \} \\
 & y{\downarrow}(j-k){\downarrow}(k-i){\uparrow}i \\
= & \quad \{\ y{\downarrow}a{\uparrow}b = y{\uparrow}(a+b){\downarrow}a \text{ , } k-i+i = k\ \} \\
 & y{\downarrow}(j-k){\uparrow}k{\downarrow}(k-i) \\
= & \quad \{\ C{\cdot}k{\cdot}j \text{ , hence: } y{\downarrow}(j-k){\uparrow}k = x{\uparrow}k\ \} \\
 & x{\uparrow}k{\downarrow}(k-i) \\
= & \quad \{\ k = i+k-i \text{ , } x{\uparrow}(a+b){\downarrow}b = x{\downarrow}b{\uparrow}a\ \} \\
 & x{\downarrow}(k-i){\uparrow}i \\
= & \quad \{\ k-i = 1+k-1-i \text{ , } x{\downarrow}(a+b) = x{\downarrow}a{\downarrow}b \text{ (heading for } k-1-i)\ \} \\
 & x{\downarrow}1{\downarrow}(k-1-i){\uparrow}i \qquad .
\end{array}
$$

We conclude that (h4) equals (h5) , which, indeed, is an instance of (z0) , albeit that y has been replaced by $x{\downarrow}1$:

(h5) $(\mathbf{MAX}\, i : 0 \leq i \leq k-1 \wedge C^{*}\cdot i\cdot(k-1) : i)$.

So, we replace $g\cdot(k-1)$ by $g\cdot(zz\cdot(k-1))$ where $zz = mpp\cdot x\cdot(x{\downarrow}1)$. The use of $zz\cdot(k-1)$ is correct, by induction hypothesis, because $k \leq j$, and so $k-1 < j+1$. This is important because the general case comprises the special case $y = x{\downarrow}1$, where we have $z = zz$.
(end •)

By application of these transformations to program1 we obtain the following program for mpp . In this program, the terms $C\cdot k\cdot j$ have been eliminated, the terms $B\cdot k\cdot j$ have been replaced by $x\cdot k = y\cdot j$, and the definition of zz has been "lifted" to the most global level possible -- at the expense of an extra name, mpx , for mpp·x -- . Lifting zz's definition is necessary for the sake of efficiency; now zz may be considered as a constant throughout the program for mpx (cf. section 6.10).

program2:
```
mpp |[ mpp·x = mpx
              |[ mpx·y = z
                        |[ z·0     = 0
                         & z·(j+1) = g·(z·j)
                                    |[ g·k = ( x·k=y·j        → k + 1
                                             [] x·k≠y·j ∧ 0=k → 0
                                             [] x·k≠y·j ∧ 0<k → g·(zz·(k-1))
                                             )
                                    ]|
                        ]|
               & zz    = mpx·(x↓1)
              ]|
    ]|
```
□

Function z in this program is not yet a list. The transformation of its definition into one that yields a list is a standard one (see section 6.4): we introduce a function f , of type $Nat \to L_\infty$, with specification $f \cdot j \cdot i = z \cdot (j+i)$. Then, we can define z by $z = f \cdot 0$, or, in order to get rid of the case analysis in z's definition, by $z = 0 \,;\, f \cdot 1$. In the definition of $z \cdot (j+1)$, j occurs only in $z \cdot j$ and $y \cdot j$; hence, using $z \cdot j = (z \downarrow j) \cdot 0$ and $y \cdot j = (y \downarrow j) \cdot 0$ -- cf. section 6.11 -- , we provide f with parameters for $z \downarrow j$ and $y \downarrow j$, thus making parameter j itself superfluous. I.e. f's specification now is:

$$(f) \qquad 0 \leqslant i \wedge 0 \leqslant j \wedge s = y \downarrow j \wedge t = z \downarrow j \;\Rightarrow\; f \cdot s \cdot t \cdot i \;=\; z \cdot (j+1+i) \qquad .$$

Thus, we obtain the final version of our first solution.

program3 (new definition of z only):

```
    z = 0 ; f·y·z |[ f·(a ; s)·(b ; t) = g·b ; f·s·t
                                            |[ g·k = ( x·k=a         → k + 1
                                                     [] x·k≠a ∧ 0=k → 0
                                                     [] x·k≠a ∧ 0<k → g·(zz·(k−1))
                                                     )
                                            ]|
                   ]|
```
□

We conclude this section with a discussion of the time complexity of program3. Evaluation of $z \uparrow (j+1)$ takes $O(j)$ time for the $\uparrow (j+1)$ operation, plus j unfoldings of function f , plus the time needed to evaluate the elements $z \cdot i$, $0 \leqslant i \leqslant j$, themselves. We have $z \cdot 0 = 0$ and $z \cdot (i+1) = g \cdot (z \cdot i)$. The form of the latter formula suggests the use of *amortized complexity*, as discussed in section 7.1, to account for the evaluations of $g \cdot (z \cdot i)$, $0 \leqslant i < j$.

We use the theory developed in section 7.1. Let $t \cdot k$ denote the number of unfoldings of g needed to evaluate $g \cdot k$; let $s \cdot k$ denote the amortized cost of $g \cdot k$ and let c be the credit function coupling s and t . I.e. c must be such that, for natural k :

$$\begin{array}{lll} & c \cdot k & \geqslant 0 \\ & c \cdot (z \cdot 0) & = 0 \quad , \quad \text{i.e.} \quad c \cdot 0 = 0 \\ (s) & s \cdot k & = t \cdot k + c \cdot (g \cdot k) - c \cdot k \qquad . \end{array}$$

In order to conclude that evaluation of the values $z{\cdot}i$, $0 \leqslant i \leqslant j$, requires $O(j)$ unfoldings of g , it suffices to show that s is $O(1)$ for some c satisfying the above requirements. From the recursive definition of g we obtain the following recurrence relations for t :

$$t{\cdot}k = 1 \qquad , \quad x{\cdot}k = a \vee 0 = k$$
$$t{\cdot}k = 1 + t{\cdot}(zz{\cdot}(k-1)) \quad , \quad x{\cdot}k \neq a \wedge 0 < k \qquad .$$

We now derive -- this derivation is due to L.A.M. Schoenmakers [Sch] -- , guided by the case analysis in g's definition:

• $x{\cdot}k = a$;

$s{\cdot}k$

$=$ { (s) , for $x{\cdot}k = a$; $t{\cdot}k = 1 \wedge g{\cdot}k = k+1$ }

$1 + c{\cdot}(k+1) - c{\cdot}k$

$\leqslant$ { assume $c{\cdot}(k+1) - c{\cdot}k \leqslant 1$ (see below) }

2 .

• $x{\cdot}k \neq a \wedge 0 = k$:

$s{\cdot}k$

$=$ { (s) , for $x{\cdot}k \neq a \wedge 0 = k$: $t{\cdot}k = 1 \wedge g{\cdot}k = 0$ }

$1 + c{\cdot}0 - c{\cdot}0$

$=$ { algebra }

1 .

• $x{\cdot}k \neq a \wedge 0 < k$;

$s{\cdot}k$

$=$ { (s) , for $x{\cdot}k \neq a \wedge 0 < k$; $t{\cdot}k = 1 + t{\cdot}(zz{\cdot}(k-1)) \wedge g{\cdot}k = g{\cdot}(zz{\cdot}(k-1))$ }

$1 + t{\cdot}(zz{\cdot}(k-1)) + c{\cdot}(g{\cdot}(zz{\cdot}(k-1))) - c{\cdot}k$

$=$ { algebra , preparing for application of (s) }

$1 + t{\cdot}(zz{\cdot}(k-1)) + c{\cdot}(g{\cdot}(zz{\cdot}(k-1))) - c{\cdot}(zz{\cdot}(k-1)) + c{\cdot}(zz{\cdot}(k-1)) - c{\cdot}k$

$=$ { (s) }

$1 + s{\cdot}(zz{\cdot}(k-1)) + c{\cdot}(zz{\cdot}(k-1)) - c{\cdot}k$

$\leqslant$ { induction hypothesis }

$$1 + 2 + c\cdot(zz\cdot(k-1)) - c\cdot k$$

$=$ { assume $c\cdot(zz\cdot(k-1)) - c\cdot k \leqslant zz\cdot(k-1) - k$ (see below) }

$$3 + zz\cdot(k-1) - k$$

$\leqslant$ { $zz\cdot(k-1) \leqslant k-1$ }

$$2 \quad .$$

(end $\cdot$)

In this derivation we have assumed that c satisfies:

$$c\cdot k - c\cdot l \leqslant k - l \quad , \quad 0 \leqslant k \wedge 0 \leqslant l \quad .$$

This and the above requirements for c are met if we choose c to be the identity function: $c\cdot k = k$ for k , $0 \leqslant k$. With this c , we now have proved:

$$s\cdot k \leqslant 2 \quad .$$

Moreover, we conclude, using (s) :

$$t\cdot k \leqslant 2 + k - g\cdot k \quad .$$

In the above analysis we have ignored the contributions of expressions $x\cdot k$ and $zz\cdot(k-1)$. The element selections $\cdot k$ and $\cdot(k-1)$ will be eliminated in the next section. The use of zz causes no problems either: because $k-1 < j+1$ we conclude that $z\uparrow(j+2)$ depends, in the worst case, on $zz\uparrow j$ only. Expression $zz\uparrow j$, therefore, has time complexity $O(j)$, and its contribution to the time complexity of $z\uparrow(j+2)$ also is $O(j)$. We conclude that program3 has linear time complexity.

10.3 The second program

By means of program transformations we eliminate the, inefficient, expressions $x\cdot k$ and $zz\cdot(k-1)$ from program3. We use the technique developed in section 6.11. We start with $zz\cdot(k-1)$.

Let $l = zz\cdot(k-1)$; if we would equip g with an additional parameter that represents $zz\cdot(k-1)$, we would need an expression for $zz\cdot(l-1)$ as argument in the recursive application of g, where we have $l < k$. Therefore, using $zz\cdot(k-1) = rev\cdot(zz\uparrow k)\cdot 0$ and $rev\cdot(zz\uparrow l) = rev\cdot(zz\uparrow k)\downarrow(k-l)$, we equip g with a parameter representing $rev\cdot(zz\uparrow k)$. We call the function thus obtained $g1$; its specification is:

(g1) $\quad 0 \leqslant k \leqslant j \wedge C\cdot k\cdot j \wedge v = rev\cdot(zz\uparrow k) \Rightarrow g1\cdot v\cdot k = g\cdot k$.

By plugging this into program3 we obtain program4.

program4:

```
    z = 0 ; f·y·z |[ f·(a;s)·(b;t) = g1·(rev·(zz↑b))·b ; f·s·t
                                      |[ g1·v·k = ( x·k=a        → k + 1
                                                 [] x·k≠a ∧ 0=k → 0
                                                 [] x·k≠a ∧ 0<k → g1·(v↓(k−l))·l
                                                                  |[ l = v·0 ]|
                                                 )
                                      ]|
                  ]|
```
□

From the previous section we know that the evaluation time of $g\cdot k$ is at most $2 + k - g\cdot k$. We now verify that the new expression $v\downarrow(k-l)$ does not disturb this. Let $t\cdot k$ be the contribution of this expression to the evaluation time of $g\cdot k$. Because $v\downarrow(k-l)$ does not occur in the first two alternatives of g's definition, we have for these cases $t\cdot k = 0$; for the case $x\cdot k \neq a \wedge 0 < k$ we derive:

$t\cdot k$

$=$ { $v\downarrow(k-l)$ takes $k-l$ time }

$k - l + t\cdot l$

$\leqslant$ { induction hypothesis }

$k - l + 2 + l - g\cdot l$

$=$ { algebra , $g\cdot l = g\cdot k$ }

$2 + k - g\cdot k$.

The elimination of expression $zz\cdot(k-1)$ from the definition of g caused the introduction of expression $rev\cdot(zz\uparrow b)$ in the application of $g1$, i.e. in the definition of f -- one level higher, so to speak -- . We now try to apply the same transformation to eliminate $rev\cdot(zz\uparrow b)$. I.e. we provide f with an additional parameter representing $rev\cdot(zz\uparrow b)$. Let $c = g\cdot b$; recalling, see (f) , that $b = z\cdot j$ and $c = z\cdot(j+1)$, we conclude that the argument supplied for this new parameter in the recursive application of f must be $rev\cdot(zz\uparrow c)$. We have $z\cdot(j+1) \leqslant z\cdot j+1$, so $c \leqslant b+1$ but not necessarily $c \leqslant b$. Hence, we cannot express $rev\cdot(zz\uparrow c)$ in terms of $rev\cdot(zz\uparrow b)$. Therefore, we use the representation of zz by the pair $[rev\cdot(zz\uparrow b), zz\downarrow b]$; then, we have:

$$[rev\cdot(zz\uparrow c), zz\downarrow c] = shift\cdot(c-b)\cdot[rev\cdot(zz\uparrow b), zz\downarrow b] \quad .$$

So, we equip f with a parameter representing this pair. Thus, we obtain function $f1$, with specification:

$$\text{(f1)} \qquad 0 \leqslant j \wedge s = y\downarrow j \wedge t = z\downarrow j \wedge b = z\cdot j \wedge z1 = [rev\cdot(zz\uparrow b), zz\downarrow b] \;\Rightarrow\; f1\cdot z1\cdot s\cdot t = f\cdot s\cdot t \ .$$

The elimination of expression $x\cdot k$ can be dealt with in exactly the same way. Observing that $x\cdot k = rev\cdot(x\uparrow(k+1))\cdot 0$ we provide $g1$ with a new parameter representing $rev\cdot(x\uparrow(k+1))$; this gives rise to the introduction, in $f1$, of a parameter representing $[rev\cdot(x\uparrow(b+1)), x\downarrow(b+1)]$. Leaving the construction of the specifications of functions $g2$ and $f2$ as an exercise to the reader, we obtain our second, and final, program. Function shift is the same as the one discussed in section 6.11. Notice that the time complexity of $shift\cdot(c-b)$ is $O(|c-b|)$, and that this does not destroy the linear time complexity of z .

program5 (again: definition of z only):

```
z = 0 ; f2·[[x·0],x↓1]·[[],zz]·y·z
    |[ f2·x1·z1·(a ; s)·(b ; t) = c ; f2·(shift·(c−b)·x1)·(shift·(c−b)·z1)·s·t
                                  |[ c          = g2·(x1·0)·(z1·0)·b
                                  & g2·u·v·k = ( u·0=a         → k + 1
                                               [] u·0≠a ∧ 0=k → 0
                                               [] u·0≠a ∧ 0<k
                                                 → g2·(u↓d)·(v↓d)·l
                                                   |[ l = v·0  &  d = k−l ]|
                                               )
                                  ]|
    ]|
```

□

10.4 Applications

With only little explanation we show a few applications of mpp . All these applications can be formulated as mpp·x·(x↓1) , for some suitable x . Therefore, we discuss mpp·x·(x↓1) first. In this section, x is a fixed, infinite list that occurs mainly as a global constant in our formulae.

10.4.0 about mpp·x·(x↓1)

We recall the informal specification of mpp from section 10.1:

mpp·x·y·j ≐ "the maximal length of any prefix of x that is a suffix of y↑j " , $0 \leqslant j$.

For the special case mpp·x·x this amounts to mpp·x·x·j = j , because x↑j itself is both a prefix of x and a suffix of x↑j . So, mpp·x·x is not very useful. We might, however, be interested in function g with specification:

g·j = "the maximal length of any prefix of x that is a *proper* suffix of x↑j " , $1 \leqslant j$.

where the proper suffixes of $x\uparrow j$ are those suffixes of $x\uparrow j$ that are shorter than j. Of course, this is meaningful for positive j only. Formalising this we derive, for $j : 0 < j$:

$$
\begin{array}{ll}
 & g\cdot j \\
= & \quad \{ \text{ formal specification of } g \ \} \\
 & (\mathbf{MAX}\, i : 0 \leqslant i < j \ \wedge \ x\uparrow i = x\downarrow(j-i)\uparrow i : i) \\
= & \quad \{ \ x\downarrow(j-i) = x\downarrow 1\downarrow(j-1-i) \ , \text{ for } i,j : i < j \ \} \\
 & (\mathbf{MAX}\, i : 0 \leqslant i \leqslant j-1 \ \wedge \ x\uparrow i = x\downarrow 1\downarrow(j-1-i)\uparrow i : i) \\
= & \quad \{ \text{ specification of mpp } \} \\
 & \text{mpp}\cdot x\cdot(x\downarrow 1)\cdot(j-1) \quad .
\end{array}
$$

Hence, we have: $g\cdot j = \text{mpp}\cdot x\cdot(x\downarrow 1)\cdot(j-1)$; conversely, $\text{mpp}\cdot x\cdot(x\downarrow 1)\cdot j$ can be interpreted as "the maximal length of any prefix of x that is a proper suffix of $x\uparrow(j+1)$" , $0 \leqslant j$.

10.4.1 periodicity computation

The *period* of a, non-empty, finite list s of length j is the least divisor i of j such that $s = (s\uparrow i)^{j/i}$. Because j is a divisor of j and because $s = (s\uparrow j)^{j/j}$, this is a correct definition. We use $i|j$ for *"i is a divisor of j"* .

property 10.4.1.0: For finite list s of length j and for i satisfying $i|j$:

$$s = (s\uparrow i)^{j/i} \ \equiv \ s\uparrow(j-i) = s\downarrow i$$

□

A formal definition of $p\cdot j$, denoting the period of $x\uparrow j$, $0 < j$, is:

$$p\cdot j \ = \ (\mathbf{MIN}\, i : 0 < i \leqslant j \ \wedge \ i|j \ \wedge \ x\uparrow(j-i) = x\uparrow j\downarrow i : i) \quad , \quad 0 < j \qquad .$$

We now derive:

$$
\begin{array}{ll}
 & p{\cdot}j \\
= & \quad \{\text{ definition of } p{\cdot}j \ \} \\
 & (\mathbf{MIN}\, i : 0 < i \leqslant j \wedge i \mid j \wedge x{\uparrow}(j-i) = x{\uparrow}j{\downarrow}i : i) \\
= & \quad \{\ x{\uparrow}j{\downarrow}i = x{\downarrow}i{\uparrow}(j-i)\ \} \\
 & (\mathbf{MIN}\, i : 0 < i \leqslant j \wedge i \mid j \wedge x{\uparrow}(j-i) = x{\downarrow}i{\uparrow}(j-i) : i) \\
= & \quad \{\text{ dummy substitution } i \leftarrow j-i \ \} \\
 & (\mathbf{MIN}\, i : 0 \leqslant i < j \wedge (j-i) \mid j \wedge x{\uparrow}i = x{\downarrow}(j-i){\uparrow}i : j-i) \\
= & \quad \{\text{ calculus }\} \\
 & j - (\mathbf{MAX}\, i : 0 \leqslant i < j \wedge (j-i) \mid j \wedge x{\uparrow}i = x{\downarrow}(j-i){\uparrow}i : i) \quad .
\end{array}
$$

The formula thus obtained resembles the specification of g , defined in the previous section, very much. The main difference is the additional conjunct $(j-i) \mid j$. With function f defined by: $f{\cdot}j = j - g{\cdot}j$, we have the following lemma.

lemma 10.4.1.0:

$$
\begin{array}{l}
f{\cdot}j \mid j \ \Rightarrow\ p{\cdot}j = f{\cdot}j \\
\neg(f{\cdot}j \mid j) \ \Rightarrow\ p{\cdot}j = j
\end{array}
$$

□

10.4.2 carré recognition

A finite list is a *carré* if it equals $s{+\!\!+}s$, for some finite list s .

property 10.4.2.0: With p the function defined in the previous section:

"$x{\uparrow}j$ is a carré" $\equiv$ $(2*p{\cdot}j) \mid j$, $0 < j$

□

10.4.3 overlap recognition

A finite list s of length j is said to *have overlap* if $(\mathrm{E}\, i : j/2 < i < j : s{\uparrow}i = s{\downarrow}(j-i))$. We have: "$x{\uparrow}j$ has overlap" $\equiv$ $g{\cdot}j > j/2$, $0 < j$.

10.4.4 pattern matching

Let s , the *pattern*, be a finite list of length k and let y , the *text*, be an infinite list; the proposition *"s occurs in y at position j"* can be formalised as $s = y{\downarrow}j{\uparrow}k$, for j , $0 \leqslant j$. Let x be an infinite list satisfying $s = x{\uparrow}k$; then, we have:

$$
\begin{array}{ll}
 & s = y{\downarrow}j{\uparrow}k \\
= & \quad \{ \; s = x{\uparrow}k \; \} \\
 & x{\uparrow}k = y{\downarrow}(k+j-k){\uparrow}k \\
\Leftarrow & \quad \{ \text{ definition of } \mathbf{MAX} \text{ ; specification of mpp } \} \\
 & \text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) = k \qquad .
\end{array}
$$

Similarly, we have: $s \neq y{\downarrow}j{\uparrow}k \Leftarrow \text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) < k$. A problem is that when $\text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) > k$ we can draw no conclusions about $s = y{\downarrow}j{\uparrow}k$. We can, however, circumvent this problem by seeing to it that $\text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) \leqslant k$. We achieve this by exploitation of the freedom we have in choosing $x{\downarrow}k$:

$$
\begin{array}{ll}
 & \text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) \leqslant k \\
\Leftarrow & \quad \{ \text{ specification of mpp , definition of } \mathbf{MAX} \; \} \\
 & (\mathbf{A}\, i : k+1 \leqslant i \leqslant k+j : x{\uparrow}i \neq y{\downarrow}(k+j-i){\uparrow}i \,) \\
= & \quad \{ \text{ dummy substitution } i \leftarrow k+1+i \; \} \\
 & (\mathbf{A}\, i : 0 \leqslant i < j : x{\uparrow}(k+1+i) \neq y{\downarrow}(j-i-1){\uparrow}(k+1+i) \,) \\
\Leftarrow & \quad \{ \; 0 \leqslant k < k+1+i : \text{Leibniz} \;\; ; \;\; (y{\downarrow}(j-i-1)){\cdot}k = y{\cdot}(j-i-1+k) \; \} \\
 & (\mathbf{A}\, i : 0 \leqslant i < j : x{\cdot}k \neq y{\cdot}(j-i-1+k) \,) \\
\Leftarrow & \quad \{ \text{ calculus } \} \\
 & (\mathbf{A}\, i : 0 \leqslant i : x{\cdot}k \neq y{\cdot}i \,) \qquad .
\end{array}
$$

So, a condition implying $\text{mpp}{\cdot}x{\cdot}y{\cdot}(k+j) \leqslant k$ is $(\mathbf{A}\, i : 0 \leqslant i : x{\cdot}k \neq y{\cdot}i)$; it is satisfied if we choose $x{\cdot}k = \surd$, where $\surd$ ("tick") is a value not occurring in y : $\surd$ is a, so-called, *sentinel*, marking the end of pattern s . If $\surd$ does not occur in s either, then x need not be an infinite list: evaluation of $\text{mpp}{\cdot}x{\cdot}y$ does not require evaluation of $x{\cdot}i$ for i ; $k < i$. Hence, we simply may take $x = s \mathbin{+\!\!+} [\surd]$ and we obtain:

$$s = y{\downarrow}j{\uparrow}k \;\equiv\; mpp{\cdot}x{\cdot}y{\cdot}(k+j) = k \quad , \quad 0 \leqslant j \qquad .$$

In this example, the recursive application mpp·x·(x↓1) , in the programs for mpp , can be interpreted as application of the pattern matching process to the pattern itself: mpp·x·(x↓1) is the preprocessing part of the Knuth-Morris-Pratt algorithm.

Preprocessing and pattern matching proper can even be combined. Observing that x↓(k+1) is irrelevant we may choose $x = s ++ [\surd] ++ y$ and use mpp·x·(x↓1) instead of mpp·x·y . We have -- observing that $y = x{\downarrow}1{\downarrow}k$ -- :

$$s = y{\downarrow}j{\uparrow}k \;\equiv\; mpp{\cdot}x{\cdot}(x{\downarrow}1){\cdot}(2{*}k+j) = k \quad , \quad 0 \leqslant j \qquad .$$

The preprocessing part now corresponds to mpp·x·(x↓1)↑(k+1) whereas the pattern matching proper corresponds to mpp·x·(x↓1)↓(k+1) : there is no real difference between the two parts.

10.5 Epilogue

We have developed a functional program with which a number of non-trivial problems can be solved in linear time. Apparently, mpp is an essential component for a whole class of programs dealing with string recognition. We discuss a few aspects of this development in isolation.

We have taken the specification of mpp for granted. When we take, however, the pattern matching problem as our starting point, the introduction of mpp is not the most obvious choice. If we are interested, for finite list s of length k and infinite list y , in the boolean values $s = y{\downarrow}j{\uparrow}k$, $0 \leqslant j$, the more obvious generalisation is: $(\mathbf{MAX}\, i : 0 \leqslant i \leqslant k \wedge s{\uparrow}i = y{\downarrow}j{\uparrow}i : i)$. If and only if this value is k we have $s = y{\downarrow}j{\uparrow}k$. A disadvantage of this expression is that it contains the length of s , which may give rise to more case analysis in the program. The advantage of mpp's specification is that it does not depend on the lengths of the lists: it is only slightly simpler than the above expression but this slight simplication turns out to be of crucial importance. The price for this, on the other hand, is the use of a sentinel to mark the end of the pattern.

The derivation of program2 is quite satisfactory. In particular, what can be considered as the crucial idea behind the Knuth-Morris-Pratt algorithm can be written down, in isolation, as a short derivation in ↑↓-calculus. Apart from this, the design of the program resembles derivations of programs for other, so-called, *segment problems*.

Program2 itself is quite nice too. The definition of z, which is the essential part, is short and simple; moreover, it has a direct sequential counterpart. As a matter of fact, program2 contains all relevant properties of the functions involved. The derivation of these properties can be, and therefore: should be, independent of the decision whether the algorithm will be coded as a functional or as a sequential program.

The most laborious part of the design is the transformation of program2, in a number of steps, into program5. This part (again) shows that program transformations can be laborious but very effective. In our case, the transformations serve to take into account several requirements regarding the use of *lists* for the representation of some of the functions involved. Thus, we have shown that the problem, and with it a whole collection of similar problems, can be solved in linear time using list operations instead of (random access) array operations. For those who care this may be a nice result.

This part of the game, however, has not much to do with functional programming; a similar sequence of transformations can be applied to the corresponding sequential program. It should be noted here that the absence of arrays in functional-program notations is usually taken for granted. The current example shows that one may have to go a long way to get rid of array operations; as a matter of fact, so long a way that it becomes questionable whether it should be done at all. Apparently, for this kind of problems solutions with arrays are to be preferred.

The transformation of program2 into program5 does not require difficult heuristics: we have only used the standard technique of introduction of additional parameters -- notice, however, that part of the work has been done already in section 6.11 -- . In a recent paper by R.S. Bird et al. [Bir2] a functional program for the KMP algorithm is presented too. In this paper the authors need to save the efficiency of their program by pulling a complicated datastructure out of the hat.

Finally, we are tempted to the conclusion that $\uparrow\downarrow$-calculus is an effective tool for calculations involving (segments of) both finite and infinite lists: it enables us to discuss lists without having to carry out the discussion in terms of their elements. Here, a careful choice of the notation used turnes out to be crucial [Gas]: only after the introduction of the $\uparrow$ and $\downarrow$ operators we discovered rules such as $x\downarrow i\uparrow j = x\uparrow(i+j)\downarrow i$.

11 Epilogue

11.0 What we have achieved

In this monograph we discussed a number of techniques for functional programming and we used them to derive, in a calculational way, programs for a number of problems. The main result of this work is that a rather small repertoire of simple techniques suffices for the systematic development of a large variety of programs. In particular, the technique of *generalisation by abstraction* fits very well in a calculational style of programming. Furthermore, a relatively simple formalism, the semantics of which is only partially defined, is sufficient for programming.

11.1 Functional programming

In some of the examples we showed how functional programs can be transformed, with little effort, into sequential programs for the same problem. Therefore, functional programming can be used in two ways, namely as a design activity in its own right, the final products of which are (executable) functional programs, or as the first phase in the development of programs to be encoded in some other program notation. In the latter case, functional programs, together with their specifications, form the starting point for what we may call the *implementation phase* of the development. We discuss this in some detail.

A (recursive) function definition in our program notation fixes the relation between the values of the function in different points of its domain. The special form of the relation enables effective computation of the function values. The relation places constraints on the order in which these values must be computed, but it does not fix this order completely. The freedom thus left can, in the implementation phase, be exploited in several ways. For instance, we may choose an order that minimises the use of storage space, or we may use parallelism for the computation of unordered values [Hoo2].

A more classical approach to the design of sequential programs is as follows. Given postcondition $x = F{\cdot}N$, for fixed natural N , we obtain, by replacement of constant N by a variable, $x = F{\cdot}n \wedge 0 \leq n \leq N$ as a tentative invariant for a program of the following form:

```
  n,x := 0 , F·0
{ invariant:  x = F·n ∧ 0 ≤ n ≤ N }
; do n ≠ N → n,x := n+1 , F·(n+1) od
{ x = F·n ∧ n = N , hence: x = F·N }  .
```

Next, we try to find suitable expressions for $F{\cdot}0$ and $F{\cdot}(n+1)$, where, because of the invariant, we are satisfied if we can express $F{\cdot}(n+1)$ in terms of $F{\cdot}n$. It remains, however, to be seen whether such expressions can be found. In this respect, the decision to use a program of the above structure must be considered as premature. Moreover, if we are, for example, able to express $F{\cdot}(2*n)$ in terms of $F{\cdot}n$ -- see chapter 4 -- , how do we exploit this in the above program?

We conclude that it is a wise strategy to derive a set of recurrence relations first and to decide upon the order of the computations later. Functional-programming techniques are well-suited for the derivation of such relations. In most cases this is the harder (and more important) part of the design.

By identification of the relations between the values to be computed we can also try to discover to what extent parallelism can be used for these computations. M. Rem has designed a number of, so-called, *systolic arrays* in a way that resembles functional programming very much [Rem]. As a result, there is a marked similarity between Rem's systolic-array programs and functional programs for the same problem. When the recurrence relations are sufficiently simple, the functional programs can be implemented quite easily as systolic arrays; when, however, these relations are not so simple, the transformation is far from trivial. This is a subject of further research.

11.2 The role of specifications

With respect to the proper role of specifications we follow C.A.R. Hoare who states [Hoa0]: "It is essential that the notations used for formalization of requirements should be mathematically meaningful, but it would be unwise to place any other restriction upon them." . A specification is the *first* formalisation of a (programming) problem. Hence, by definition, we cannot speak of the correctness, in the mathematical sense of the word, of a specification; we can only convince ourselves that a specification specifies the right thing by interpreting its meaning, in an informal way. Good specifications are, therefore, as self-evident as possible.

Furthermore, a good specification defines only those properties of the specified object that we are interested in, and nothing else. Mathematically, this means that specifications should be as weak as possible. This leaves us a maximal freedom for the construction of a program: the weaker a specification is, the more programs satisfy it. In practice, some overspecification cannot always be avoided, but we still should try to avoid it as much as possible.

On account of the above, we do not share the opinion that functional programs are well-suited to be used as, so-called, *executable specifications*. First, because programs contain information on *how* the specified objects can be computed, programs are, by definition, always overspecific, to an extent that must be considered as undesirable: this may impose such a bias onto the design that it becomes virtually impossible to derive algorithms that are not refinements of the algorithm denoted by the specification. Second, the use of a program notation does not always yield the simplicity and clarity needed to obtain the required self-evidentness of the specification. Examples of non-executable specifications can be found in chapters 6 , 7 , 8 , and 9. These examples show that equations, specifying the objects implicitly, sometimes are clearer than explicit definitions of these objects. This is particularly the case with representations of abstract datatypes: such representations are specified in terms of the abstract values they represent. (Examples: 6.5 , 6.7 , 6.8 , 7 , and 9 .)

11.3 The equivalence of computations

A program is meaningful only with respect to a specification. For a given specification two programs may be considered as equivalent, with respect to that specification, if they both do or do not satisfy it. A consequence of this attitude is that there is no point in discussing the equivalence of two programs without taking into account their (common) specification.

We illustrate this with an example. We consider definitions of functions f and g of the following form:

```
f·x  =  ( 0 ⩽ x → F )
g·x  =  ( 0 ⩽ x → G )  .
```

Since these definitions provide no information whatsoever about the values of f·x and g·x for non-natural values of x , it is reasonable to assume that such definitions can only have been derived from specifications stating properties of the function for natural arguments only. By constructing such a specification we express our deliberate decision not to be interested in f·x for non-natural x . With respect to this specification, f and g are equivalent if (Ax : 0 ⩽ x : f·x = g·x) . Once we have taken this decision, we should not, then, care about whether or not f·(−3) = g·(−3) : within the context of our specification this question is irrelevant. Similarly, with respect to this specification, function f is also equivalent to function h defined by:

```
h·x  =  ( 0 ⩽ x → F
        ▯ x < 0 → h·(x−1)
        )  .
```

Although both f·(−3) and h·(−3) are "undefined" , we can think of an implementation in which evaluation of f·(−3) terminates whereas evaluation of h·(−3) does not: now, should these expressions be considered as equal or not? As before, the question is irrelevant.

This example shows two things. First, in order to answer questions regarding undefined values, rather complicated denotational models for the program notation are needed. The above shows that this is unnecessary: values are left undefined deliberately, because we are not interested in them. We do not even need special names, such as $\perp$, for undefined values. Second, the above exposes a serious drawback of program transformation systems. In such a system, programs are transformed into equivalent programs, without taking into account the specification of the program one starts with. As a result, the only thing one can do in such a system is to prove the equivalence of the programs in the strongest sense of the word. In the above example, the proof obligation would be (Ax : Ω·x : f·x = g·x) , which is unnecessarily hard.

References

[Bir0] R.S. Bird, P. Wadler
Introduction to Functional Programming
Prentice Hall International, Hemel Hempstead, 1988.

[Bir1] R.S. Bird
Lectures on Constructive Functional Programming
Oxford University Computing Laboratory, PRG note 69, 1988.

[Bir2] R.S. Bird, J. Gibbons, G. Jones
Formal Derivation of a Pattern Matching Algorithm
Science of Computer Programming 12, 1989, pp. 93-104.

[Boo] P. Boots
Twee generalisaties van het begrip productiviteit (in Dutch)
private communication, Eindhoven, 1986.

[Bur] R.M. Burstall, J. Darlington
A Transformation System for Developing Recursive Programs
Journal of the ACM 24, 1977, pp. 44-67.

[Coo] D.C. Cooper
The equivalence of certain computations
Computing Journal 9, 1966, pp. 45-52.

[Dar0] J. Darlington
Program Transformation
in [Dar1].

[Dar1] J. Darlington, P. Henderson, D.A. Turner (eds.)
Functional Programming and its Applications
Cambridge University Press, Cambridge, 1982.

[Dij0] E.W. Dijkstra
A Discipline of Programming
Prentice-Hall, Englewood Cliffs, 1976.

[Dij1] E.W. Dijkstra
On the productivity of recursive definitions
EWD749, Nuenen, 1980.

[Dij2] E.W. Dijkstra
Hamming's exercise in SASL
EWD792, Nuenen, 1981.

[Dij3] E.W. Dijkstra, W.H.J. Feijen
A Method of Programming
Addison-Wesley Publishing Company, Reading Massachusetts, 1988.

[Dij4] E.W. Dijkstra, W.H.J. Feijen
The Linear Search Revisited
Structured Programming 10, 1989, pp. 5-9.

[Gas] A.J.M. van Gasteren
On the shape of mathematical arguments
Ph. D. thesis, Eindhoven University of Technology, 1988.

[Gri] D. Gries
The Science of Programming
Springer-Verlag, New York, 1981.

[Hen] M.C. Henson
Elements of Functional Languages
Blackwell Scientific Publications, Oxford, 1987.

[Hin] J.R. Hindley, J.P. Seldin
Introduction to Combinators and λ-Calculus
Cambridge University Press, Cambridge, 1986.

[Hoa0] C.A.R. Hoare
An Overview of Some Formal Methods for Program Design
IEEE Computer, September 1987, pp. 85–91.

[Hoa1] C.A.R. Hoare
Communicating Sequential Processes
Prentice Hall International, London, 1985.

[Hoo0] R.R. Hoogerwoord
On degrees of productivity
memorandum rh75, Eindhoven, 1985.

[Hoo1] R.R. Hoogerwoord
A simple theorem and its applications
memorandum rh82a, Eindhoven, 1986.

[Hoo2] R.R. Hoogerwoord
On maximal common segments of two sequences
memorandum rh85, Eindhoven, 1986.

[Knu] D.E. Knuth, J.H. Morris, V.R. Pratt
Fast Pattern Matching in Strings
SIAM Journal on Computing 6, 1977, pp. 323–350.

[Pey] S.L. Peyton Jones
The Implementation of Functional Programming Languages
Prentice Hall International, Hemel Hempstead, 1987.

[Rem] M. Rem
Small programming exercises 18, 19
Science of Computer Programming 9, 1987, pp. 91–100 and 207–211.
Small programming exercises 20
Science of Computer Programming 10, 1988, pp. 99–105.

[Sch] L.A.M. Schoenmakers
private communication, Eindhoven, 1989.

[Sij] B.A. Sijtsma
Verification and Derivation of Infinite-List Programs
Ph. D. thesis, Rijksuniversiteit Groningen, 1988.

[Tar] R.E. Tarjan
Amortized Computational Complexity
SIAM Journal on Algebraic and Discrete Methods 6, 1985, pp. 306-318.

[Tur0] D.A. Turner
SASL Language Manual
University of St. Andrews, 1976.

[Tur1] D.A. Turner
A New Implementation Technique for Applicative Languages
Software-Practice and Experience 9, 1979, pp. 31-49.

[Tur2] D.A. Turner
Recursion Equations as a Programming Language
in [Dar1].

Index

Samenvatting

Het is inmiddels algemeen bekend dat de correctheid van computerprogramma's alleen kan worden aangetoond door middel van wiskundige bewijzen. Dit gegeven laat twee manieren van programmeren toe. Enerzijds kan men eerst een programma ontwerpen om vervolgens te bewijzen dat het correct is. Anderzijds kan men het gegeven dat de correctheid van het programma moet worden bewezen als uitgangspunt nemen door programma en correctheidsbewijs tegelijkertijd te ontwerpen. Het (heuristische) voordeel van deze benadering is dat de bewijsverplichting nu als leidraad voor het ontwerp fungeert.

De laatste benadering blijkt zeer effectief te zijn. Het gelijktijdig ontwerpen van programma en correctheidsbewijs wordt *afleiden* genoemd en het wiskundig betoog aan de hand waarvan de programmacode wordt geconstrueerd een *afleiding*. Mits voldoende zorgvuldig geformuleerd vormt de afleiding tevens het correctheidsbewijs. De ervaring leert dat afleidingen ten minste een ordegrootte langer zijn dan de aldus afgeleide programma's. De hieruit voortvloeiende wens afleidingen zo compact mogelijk, maar toch voldoende gedetailleerd, te noteren heeft in de laatste 10 jaar geleid tot een *calculationele wijze* van programmeren: programma's worden door middel van formulemanipulatie uit hun specificaties afgeleid.

Het doel van het aan dit proefschrift ten grondslag liggende onderzoek was technieken te ontwikkelen voor het op calculationele wijze afleiden van functionele programma's. Het idee was dat dit niet al te moeilijk zou moeten zijn, omdat functionele-programmanotaties meer dan notaties voor sequentiële programma's op "gewone" wiskundige formalismen lijken. Verder vroegen wij ons af in welke mate functioneel programmeren van sequentieel programmeren — ook wel *imperatief programmeren* genoemd — verschilt.

Dit proefschrift gaat over het afleiden van functionele programma's; het gaat niet over functionele programmeertalen noch over implementaties hiervan. De in dit proefschrift gedefinieerde programmanotatie is niet het onderwerp van de studie, maar een (onmisbaar) middel tot een doel, namelijk programmeren. Voor dit doel volstaat het de gebruikte notatie axiomatisch te definiëren: volledigheid is van ondergeschikt belang, zolang de axioma's toereikend zijn voor het spelen van een interessant programmeerspel.

Dit proefschrift bestaat uit drie delen. In het eerste deel -- hoofdstukken 1, 2, 3 en 5 -- wordt een programmanotatie ingevoerd en wordt enige theorie voor het gebruik ervan geformuleerd. De notatie is geïnspireerd door SASL maar is gedurende het onderzoek voortdurend aan de zich ontwikkelende behoeften aangepast. De belangrijkste moraal van dit deel is dat de benodigde theorie tamelijk eenvoudig is.

In het tweede deel -- hoofdstukken 4 en 6 -- worden een aantal programmeertechnieken behandeld. De technieken in hoofdstuk 4 hebben betrekking op recursie, generalisatie, tupelvorming en het gebruik van extra parameters. Deze technieken zijn elementair: ze zijn eenvoudig en vrijwel altijd toepasbaar. De techniek *generalisatie door abstractie* blijkt goed te passen bij een calculationele programmeerwijze: door formele manipulatie worden vaak expressies verkregen die geringe (maar essentiële) verschillen vertonen; door van deze verschillen te abstraheren kan een bruikbare generalisatie van het probleem worden verkregen. Hoofdstuk 6 bevat technieken voor en voorbeelden van programma's waarin *lijsten* een rol spelen. Deze technieken worden afgeleid met behulp van de technieken uit hoofdstuk 4 en de theorie uit hoofdstuk 5.

In het derde deel -- hoofdstukken 7, 8, 9 en 10 -- passen we de technieken uit het tweede deel toe op een viertal programmeerproblemen, waaronder een combinatorisch probleem en een patroonherkenningsprobleem.

Het belangrijkste resultaat van het verrichte onderzoek is dat een betrekkelijk klein repertoire van eenvoudige technieken toereikend is voor het op calculationele wijze afleiden van functionele programma's. De afleidingen in, bij voorbeeld, hoofdstukken 4 en 9 tonen bovendien aan dat verschillende programma's voor hetzelfde probleem met deze technieken vaak met geringe extra moeite kunnen worden verkregen.

Bij het ontwerpen van sequentiële programma's kunnen we onderscheid maken tussen het karakteriseren van de uit te rekenen waarden, en de relaties daartussen, enerzijds en het kiezen van een volgorde waarin die waarden zullen worden uitgerekend anderzijds. Voor het afleiden van deze relaties kunnen functionele programmeringstechnieken met vrucht worden gebruikt; aldus kan functioneel programmeren bijdragen aan een betere verkaveling van de afleidingen van sequentiële programma's.

Er zijn aanwijzingen dat functioneel programmeren ook kan worden gebruikt bij het ontwerpen van, zogenaamde, *systolische arrays*. Dit is een onderwerp van verder onderzoek.

Curriculum vitae

9 augustus 1952	: geboren te Gorinchem
12 juni 1970	: eindexamen Gymnasium β, Gymnasium Camphusianum, Gorinchem
september 1970	: aanvang van de studie elektrotechniek, later voortgezet in de wiskunde, Technische Hogeschool Eindhoven
mei 1974 - april 1979	: studentassistent bij het rekencentrum, Technische Hogeschool Eindhoven
12 oktober 1979	: doctoraal examen wiskunde, met lof, bij prof. dr. E.W. Dijkstra, Technische Hogeschool Eindhoven; titel van het afstudeerverslag: *On the design of instruction codes.*
oktober 1979 - november 1981	: wetenschappelijk assistent in de vakgroep informatica, onderafdeling der Wiskunde en Informatica, Technische Hogeschool Eindhoven
december 1981 - augustus 1984	: medewerker ontwikkeling bij de hoofd-industriegroep Science and Industry, Nederlandse Philipsbedrijven B.V.
september 1984 - heden	: universitair docent in de vakgroep informatica, faculteit der Wiskunde en Informatica, Technische Universiteit Eindhoven

Stellingen

behorend bij het proefschrift

The design of functional programs: a calculational approach

van

Rob Hoogerwoord

Eindhoven,
19 december 1989

0. Zij C een verzameling en $<$ een partiële ordening op C zodanig dat $(C, <)$ well-founded is. Dan geldt voor alle verzamelingen V, functies $t : V \to C$ en predicaten $P : V \to Bool$ (met $x \in V \wedge y \in V$):

$$(\mathbf{A}x :: P \cdot x) \Leftarrow (\mathbf{A}x :: P \cdot x \Leftarrow (\mathbf{A}y : t \cdot y < t \cdot x : P \cdot y)) \quad .$$

1. Met behulp van de vorige stelling kan een aanzienlijk eenvoudiger en korter bewijs van de invariantiestelling worden afgeleid dan dat van A.J.M. van Gasteren.

   A.J.M. van Gasteren
   On the shape of mathematical arguments
   proefschrift, Technische Universiteit Eindhoven, 1988.

2. Voor het bewijzen van eigenschappen van (functies op) recursieve datatypen is structurele inductie in het algemeen ontoereikend.

   R.S. Bird, P. Wadler
   Introduction to functional programming
   Prentice Hall International, Hemel Hempstead, 1988.

3. De in hoofdstuk 5 van [0] gegeven voorbeelden, ter motivering van de noodzaak van op domeintheorie gebaseerde modellen voor de λ-calculus, zijn misleidend en niet ter zake.

   [0] J.E. Stoy
   Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory
   The MIT Press, Cambridge Massachusetts, 1977.

4. De in dit proefschrift behandelde productiviteitsstelling voor oneindige lijsten vertoont gelijkenis met de contractiestelling van Banach voor volledige metrische ruimten.

5. Het ontwerpen van een betere programmeertaal heeft alleen zin wanneer hieraan een betere wijze van programmeren ten grondslag ligt, in enige zin van het woord "betere".

6. In informaticacurricula dient expliciet aandacht te worden besteed aan het ontwerpen van formele specificaties.

7. (Natuurlijke) taal is een artefact en derhalve voor verbetering vatbaar. Prescriptieve taalkunde heeft dan ook bestaansrecht.

8. De Von Neumann machine is zo gek nog niet.