Programming by Calculation: techniques and applications

DRAFT

version: 15 april 2013

Rob R. Hoogerwoord

1 november 1997

Contents

Chapter 0

Introduction

To be able to practise calculational programming in an effective and efficient way, we must pay some attention to the mathematics used. This chapter is about notation and conventions, and it introduces the calculational style.

Chapter 1

Basic Functional Programming

1.0 The Applicative Algebra

In this chapter we introduce a very simple, and yet far from trivial, formalism, which we call the *applicative algebra*. Actually, this algebra just is the λ-calculus, cast in a form more suited for functional programming. (We return to this in Section 1.3.) It models function application, and nothing else: this formalism is at the heart of every functional programming language, and we shall use it to explain a few important rules of the game.

In addition, because it is simple and abstract, the applicative algebra is a well-suited vehicle to become familiar with (uninterpreted) calculational reasoning. Here "abstract" means: without any meaning of its own.

A functional program is an expression in a (functional) language, and the meaning of such an expression is its value. As a very simple example, $2+3$ is an expression, and 5 is its value. Actually, 5 itself is an expression too, also having 5 for its value. If we would be very strict we would have to distinguish the expression "5" from its value "five" (say), but for our purposes such strictness usually is not required. In this chapter, however, the distinction is (somewhat) relevant.

1.0.0 the value domain Ω

The set of all possible values of functional expressions is called Ω .Throughout this chapter all variables have type Ω; therefore, in this chapter we shall mostly omit the type denotations (like $x{\in}\Omega$) from our formulae.

In Ω we have a binary operator $\bullet$ ("fat dot"), so that we can form expressions over Ω like $x \bullet y$, $x \bullet (y \bullet z)$, $(x \bullet y) \bullet z$. To reduce the use of parentheses we adopt the following parsing convention:

- $\bullet$ is left-binding, that is, $x \bullet y \bullet z$ is read as $(x \bullet y) \bullet z$.

For the time being, operator $\bullet$ is not the same as ordinary function application, as denoted by $\cdot$, but later we will identify them, as they will play the same role: eventually, $\bullet$ will just be (the representation of) $\cdot$ in the functional language.

1.0.1 the Dot Axiom

We wish to be able to define function-like objects in the applicative algebra. For that purpose we consider equations of the following shape, in which E is an expression:

(0) $\quad x : \ (\forall y, z :: x \bullet y \bullet z = E)$

We call equations of this shape *admissible*. In (0) x is called the *unknown* of the equation, y and z are called x's *parameters*, and E is a well-formed expression composed from variables and $\bullet$'s; expression E is called x's *defining expression*. Among the variables occurring in E may be x itself and the parameters y, z. If x does occur in E the equation is called *recursive*. All other variables occurring in E are *free*; free variables are used to represent values that, for the sake of modularity, have been defined "elsewhere".

In formula (0) the unknown x has *two* parameters, but an admissible equation may have *any* number of parameters, even zero. The central rule of the applicative algebra is that every admissible equation has (at least one) solutions in Ω. The following axiom expresses this for an equation with 2 parameters, and similar versions of the axiom can be formulated for any other number of parameters.

axiom 0 (Dot Axiom) : For every expression E over Ω:

$$(\exists x : x \in \Omega : \ (\forall y, z :: x \bullet y \bullet z = E))$$

□

example 1 : Here are a few admissible equations:

(0 parameters) : $\quad x : \ x = x$
(0 parameters) : $\quad x : \ x = x \bullet x$

(1 parameter) : $x:\ (\forall y :: x \bullet y = y\,)$
(2 parameters): $x:\ (\forall y, z :: x \bullet y \bullet z = y\,)$
(2 parameters): $x:\ (\forall y, z :: x \bullet y \bullet z = z \bullet x \bullet y\,)$
(3 parameters): $c:\ (\forall f, g, x :: c \bullet f \bullet g \bullet x = f \bullet (g \bullet x)\,)$

□

1.0.2 declarations

A *definition*, in the ordinary mathematical meaning of the word, introduces a mathematical object having properties specified in that definition; to be a proper definition, it must be clear that the object thus defined really exists.

In the applicative algebra, a *declaration* is a definition of a restricted shape, that defines a value in Ω. A declaration is obtained by writing an admissible equation in abbreviated form. For example, the admissible equation:

$$x:\ (\forall y, z :: x \bullet y \bullet z = E\,)$$

yields the declaration:

$$x \bullet y \bullet z \;=\; E \quad .$$

This declaration defines x to be a solution of the corresponding admissible equation, so the one-and-only property of x thus specified is its being a solution of the equation. The existence of such a solution is guaranteed by the Dot Axiom, so declarations are proper definitions indeed.

The admissible equation can be reconstructed from the declaration abbreviating it, because all admissible equations have the same shape: the unknown of the equation always is the left-most variable in the declaration, and the equation always is universally quantified over the unknown's parameters.

It may very well be that an admissible equation has many solutions and one might wonder which of these many solutions is bound to the variable in the declaration. The answer is that we leave this unspecified, for the simple reason that we do not care: whenever we formulate a declaration, all we are interested in is that the variable thus defined solves the corresponding equation; hence, from this point of view all solutions are equivalent, so we even should not care! (If we would care, we should have formulated a stronger, more restrictive, equation in the first place.) In Chapter 4 we shall explore the consequences of this attitude in greater detail.

1.0.3 instantiation

Because a declaration implies universal quantification over the parameters, these parameters are just dummies –bound variables– of that declaration.

Therefore, renaming of parameters is always possible (and sometimes even necessary). For example, the declaration:

$$x \bullet y \bullet z \;=\; E \quad ,$$

is equivalent with the following one, for *fresh* names u and v (which means that they do not occur in E):

$$x \bullet u \bullet v \;=\; E(y{:=}u)(z{:=}v) \quad .$$

The following rule tells us how variables defined in a declaration may be used; this rule is obtained, simply by instantiation of the corresponding admissible equation.

property 2 (Rule of Instantiation): If x is declared as follows:

$$x \bullet y \bullet z \;=\; E \quad ,$$

then for all expressions A, B, not containing y, z as free variables, x satisfies:

$$x \bullet A \bullet B \;=\; E(y{:=}A)(z{:=}B) \quad .$$

□

The condition, that the expressions substituted for the parameters do not contain these parameters as free variables, is necessary to avoid name clashes; this condition can always be met, namely by renaming the parameters first.

In calculational derivations we shall record applications of the Rule of Instantiation to a declared variable x by means of the hint "declaration of x", or (if no confusion is possible) simply by "x".

The Rule of Instantiation states an equality, for example between $x \bullet A \bullet B$ and $E(y{:=}A)(z{:=}B)$, and because equality is symmetric, we can exploit this equality in either direction: we may replace a (sub)expression $x \bullet A \bullet B$ by $E(y{:=}A)(z{:=}B)$ but we may equally well replace $E(y{:=}A)(z{:=}B)$ by $x \bullet A \bullet B$: the equalities in declarations (and their admissible equations) are true mathematical equalities.

1.0.4 where-clauses

Several declarations can be grouped together into a single new declaration by connecting them with the symbol & ("and"). Such a *multiple declaration* provides a simultaneous definition for many variables. The textual order in

which declarations are grouped together is irrelevant and *mutual recursion* is permitted; that is, the declared variables may mutually occur in each other's defining expressions.

By embracing a declaration with $\mathsf{whr} \cdots \mathsf{end}$ we obtain a so-called *where-clause*. A where-clause serves to bind a declaration to an expression, just by post-fixing that expression with the where-clause. That is, from an expression E and a (possibly multiple) declaration D we obtain as a new expression:

$$E \ \mathsf{whr}\ D\ \mathsf{end} \quad .$$

The scope of the variables declared in a where-clause comprises both the expression preceding the where-clause and all defining expressions within the where-clause. That is, when we are proving properties (or are performing program transformations) we may apply the Rule of Instantiation to any occurrence of a variable declared in that where-clause, both in the preceding expression and in the where-clause itself.

example: The following declaration defines a variable fib; the defining expression for fib contains a where-clause with a multiple declaration for (local) variables s and t; the defining expression for s contains t and the defining expression for t contains s: these declarations are mutually recursive. Also, fib's declaration itself is recursive: fib occurs in the declaration of t, which is part of fib's defining expression. Finally, this declaration contains free variables cons, zero, one, and add.

$$\begin{array}{lll} \mathit{fib} & = & \mathit{cons} \bullet \mathit{zero} \bullet s \\ & & \mathsf{whr}\ \ s = \mathit{cons} \bullet \mathit{one} \bullet t \\ & & \ \ \&\ \ t = \mathit{add} \bullet \mathit{fib} \bullet s \\ & & \mathsf{end} \end{array}$$

□

1.0.5 free variables

A functional program is an expression in the functional language, without free variables: every variable in a functional program is either a parameter or a declared variable, and, thus, is bound. Free variables only arise when we study a subexpression in isolation, by temporarily ignoring the larger expression it is part of. This happens when we study a defining expression in isolation, in which case the parameters become free, and when we study an expression outside the context of its "surrounding" declarations, in which case the declared

variables become free.

1.0.6 the Anti-Singleton Axiom

The value domain Ω is non-empty, because we can think up infinitely many admissible equations and, by the Dot Axiom, every admissible equation has a solution in Ω.

The Dot Axiom, however, does not require that different equations have different solutions. In other words, the Dot Axiom admits the possibility that all these solutions be equal, in which case Ω would be a singleton and useless. The only way to prevent this is by excluding it by means of an additional axiom.

axiom 3 (Anti-Singleton):

$$(\forall x : x{\in}\Omega : (\exists y : y{\in}\Omega : x{\neq}y))$$

□

In all its simplicity this axiom is very important because it is the only rule of our game that contains a $\neq$ symbol: all other rules express equalities. Hence, the Anti-Singleton Axiom plays a key role whenever we wish to prove that two values are different.

The simplest possible admissible equation is $x : x = x$, and it has a solution in Ω. We do not know anything about this solution, but it is the simplest example showing that Ω is non-empty.

So, we now have $(\exists x : x{\in}\Omega : \mathsf{true})$, and as a result we also obtain:

$$
\begin{array}{ll}
 & (\exists x, y :: x \neq y) \\
\equiv & \quad \{ \text{ nesting dummies } \} \\
 & (\exists x :: (\exists y :: x \neq y)) \\
\equiv & \quad \{ \text{ Anti-Singleton Axiom } \} \\
 & (\exists x :: \mathsf{true}) \\
\equiv & \quad \{ \ \Omega \text{ is non-empty } \} \\
 & \mathsf{true} \ .
\end{array}
$$

Thus we obtain a useful variation on the Anti-Singleton Axiom, stating that Ω contains at least two (different) elements.

property 4 :

$$(\exists x, y : x, y \in \Omega : x \neq y\ )$$

□

Once we have established some differences, we can obtain more by means of Leibniz's rule, as follows. In the previous chapter we have formulated Leibniz's rule in these two ways:

$$\begin{array}{lll} x = y & \Rightarrow & (\forall f :: f \cdot x = f \cdot y\ ) \quad , \text{ and} \\ f = g & \Rightarrow & (\forall x :: f \cdot x = g \cdot x\ ) \quad . \end{array}$$

By contraposition we obtain versions that can be used to derive differences from other differences:

$$\begin{array}{lll} x \neq y & \Leftarrow & (\exists f :: f \cdot x \neq f \cdot y\ ) \quad , \text{ and} \\ f \neq g & \Leftarrow & (\exists x :: f \cdot x \neq g \cdot x\ ) \quad . \end{array}$$

In words, the first rule states that two values are different if some function applied to these values yields different results; the latter rule states that two functions are different if they have different values in at least one point of their (common) domain.

So, for all practical purposes, Leibniz's rule assumes four different shapes, each of which we shall refer to by the phrase "Leibniz". (See Section 1.2 for examples.)

1.1 Functions and Types

1.1.0 connecting the dots

For any $f, f \in \Omega$, we can define a function F, of type $\Omega \to \Omega$, by:

$$(\forall x :: F \cdot x\ =\ f \bullet x) \quad .$$

Because F is completely determined by f, we say that F, which is not in Ω, is *represented in* Ω *by* f; in this terminology we also have that $\cdot$ is represented in Ω by $\bullet$, because of the equality $F \cdot x = f \bullet x$, for all x.

So, although the elements of Ω are not functions we can view and use them as functions; in addition, the shape of the admissible equations, and the Rule of Instantiation derived from it, have been chosen to comply with this view. For example, the declaration:

$$f \bullet x\ =\ E \quad ,$$

defines a "function" f and, for this f, the Rule of Instantiation reads, for any expression A:

$$f \bullet A \;=\; E(x{:=}A) \quad ,$$

which provides the necessary rule to determine the value of function f in point A of its domain.

Because the one-and-only purpose of operator $\bullet$ is to represent function application in Ω, there is no need to distinguish $\bullet$ and $\cdot$ anymore; therefore, from here onwards we identify the two and we will use one and the same symbol, $\cdot$, for both. In the same vein, whenever we declare a variable f by:

$$f \cdot x \;=\; E \quad ,$$

we will simply speak of "function f" instead of "the function represented by f". Not surprisingly, we call a declaration of this shape a *function declaration*.

Thus, we call declared variables with one (or more) parameters *functions* because they will be used as functions. Similarly, following mathematical tradition, we call declared variables with zero parameters *constants*.

1.1.1 function types

The concept of *type* is important, both in mathematics and in computing science, and much research effort has been devoted to the development of syntactic theories of types. Fortunately, for our purpose –systematic program construction– , we can get away with a very simple attitude towards types: we just treat them semantically.

Informally, a type is a common property of a collection of values; in the applicative algebra, "having a common property" can be modelled conveniently as "being an element of the same subset". Therefore, in the applicative algebra, a *type* just is a subset of Ω.

definition 5: A *type* is a subset of Ω.
□

A *function type* is the common property of mapping all elements of one set –the function's *domain*– to the elements of one and the same other set –the function's *range*– .

definition 6: For types X and V, the *function type* $X \to V$ is the subset of Ω defined by:

$$f \in (X \to V) \quad \equiv \quad (\forall x : x \in X : f \cdot x \in V) \quad , \text{ for all } f \text{ in } \Omega \quad .$$

□

In a function type $X \to V$, type X is called the *domain* of the functions in that type, and V is called their *range*. For a given function, its domain and range are not unique, because the function is an element of very many function types; for example, function types have the following monotonicity properties, for function f and types X, Y, V, W:

$$\begin{array}{lcl} X \subseteq Y & \Rightarrow & (\, f \in (Y \to V) \;\Rightarrow\; f \in (X \to V)\,) \\ V \subseteq W & \Rightarrow & (\, f \in (X \to V) \;\Rightarrow\; f \in (X \to W)\,) \end{array}$$

simple example: In this view, every function on the integers also is a function on the naturals, and every natural-valued function also is an integer-valued function.

□

1.1.2 function equivalence

Two functions f and g, both of type $\Omega \to \Omega$, are *equivalent* whenever they satisfy:

$$(\forall x :: f \cdot x = g \cdot x) \quad .$$

Equivalent functions need not be equal; yet, there is no way in which they can be distinguished, and to all intents and purposes, therefore, can they be considered the same function.

The situation becomes slightly more complicated when we consider function types. If, for some function type $X \to V$, functions f and g satisfy:

$$(\forall x : x \in X : f \cdot x = g \cdot x) \quad ,$$

then f and g are *equivalent with respect to* their domain X; outside X they may have different values, but if we are only interested in f and g on domain X then, again, with respect to this domain they can be considered the same.

example: Consider functions f and g, of type $\mathsf{Int} \to \mathsf{Int}$, defined as follows, for all integer x:

$$\begin{array}{lcll} f \cdot x & = & 3 & , \text{ if } x \le 0 \;, \\ f \cdot x & = & f \cdot (x-1) + 2 & , \text{ if } x > 0 \;, \\ g \cdot x & = & 2 * x + 3 & . \end{array}$$

On their domain, Int, these functions are different, because, for example, we have $f\cdot(-1) \neq g\cdot(-1)$. If, however, we only use f and g as functions of type $\mathsf{Nat}\to\mathsf{Nat}$ then they are equivalent and, therefore, within $\mathsf{Nat}\to\mathsf{Nat}$ they may be considered equal.

□

It is important, though, to stay aware of the distinction between functions and function declarations. Different function declarations may define equivalent functions, but in other respects these declarations may have different properties. (In the above example, for instance, the declaration of f is recursive, whereas the declaration of g is not.) This becomes particularly relevant when we study the *efficiency* of functional programs: very often, the same function can be defined by means of different declarations, with (very) differing time or space complexities. We will discuss this more extensively in a later chapter.

1.1.3 function composition

For function composition we use the infix-operator $\circ$: for any two functions f, g, of type $\Omega\to\Omega$, their composition, also of type $\Omega\to\Omega$, is denoted by $f\circ g$; as usual, we have, for all x:

$$(f\circ g)\cdot x \;=\; f\cdot(g\cdot x) \quad .$$

Function composition is associative, that is, for all f, g, h:

$$(f\circ g)\circ h \;=\; f\circ(g\circ h) \quad .$$

If functions f, g have been declared as elements of Ω then their composition $f\circ g$ is (representable) in Ω as well, because here is a valid declaration for a function h that equals $f\circ g$ –also see example 1–:

$$h\cdot x \;=\; f\cdot(g\cdot x) \quad .$$

1.1.4 functions with many parameters

In very much the same way, a declaration of the shape:

$$f\cdot x\cdot y \;=\; E \quad ,$$

can be viewed as a declaration of a two-argument function f; this function has type $\Omega\to(\Omega\to\Omega)$. As before, the Rule of Instantiation provides the necessary rule to determine the value of this function in points A and B of its domain:

$$f\bullet A\bullet B \;=\; E(y:=A)(z:=B) \quad .$$

1.2 Examples and Exercises

In all its simplicity, the applicative algebra is rich enough to be useful as a functional programming language. All extensions to be introduced in later chapters can be implemented in the applicative algebra and can thus be viewed as "syntactic sugar" only. To illustrate this, we give a few examples showing how some of the elementary datatypes can be represented in the algebra. In addition we will see how properties of functions and constants declared in the applicative algebra can be derived in a calculational way.

1.2.0 a few combinators

We consider the so-called *combinators* $I, K,$ and C, declared by:

$$\begin{array}{rrcl} & I\cdot x & = & x \\ \& & K\cdot x\cdot y & = & x \\ \& & C\cdot x\cdot y & = & y \end{array}$$

In words, I is the identity function that simply returns its argument as its value; it has type $X \to X$, for every type X. The two-argument function K returns its first argument, whereas C returns its second argument.

As an example of how we can derive properties of declared variables, we prove here that I and K are different:

$$\begin{array}{ll} & I \neq K \\ \Leftarrow & \quad \{ \text{ Leibniz, to enable use of the Rule of Instantiation } \} \\ & (\exists x :: I\cdot x \neq K\cdot x) \\ \equiv & \quad \{ \text{ declaration of } I \text{ } \} \\ & (\exists x :: x \neq K\cdot x) \\ \Leftarrow & \quad \{ \text{ Leibniz, for the same reason } \} \\ & (\exists x :: (\exists y :: x\cdot y \neq K\cdot x\cdot y)) \\ \equiv & \quad \{ \text{ declaration of } K \text{ } \} \\ & (\exists x :: (\exists y :: x\cdot y \neq x)) \\ \Leftarrow & \quad \{ \text{ instantiation } x := I, \text{ (the simplest way) to get rid of } x\cdot y \text{ } \} \\ & (\exists y :: I\cdot y \neq I) \\ \equiv & \quad \{ \text{ declaration of } I \text{ } \} \\ & (\exists y :: y \neq I) \\ \equiv & \quad \{ \text{ Anti-Singleton Axiom } \} \end{array}$$

$$\mathsf{true} \quad .$$

1.2.1 pairs

Perhaps the simplest possible data structure is a *pair* of values, viewed as a single entity; the two values making up a pair are called its *elements*. Besides a function for pair formation we also need functions to "extract" the elements from a pair. More precisely, this means that we need three functions P, L, and R, say, with the following properties. These properties specify that $P{\cdot}x{\cdot}y$ is a pair with elements x and y, and that $L{\cdot}p$ and $R{\cdot}p$ yield the left and right elements of pair p.

specification: for all x, y:

$$\begin{array}{lcl} L{\cdot}(P{\cdot}x{\cdot}y) & = & x \quad , \text{ and} \\ R{\cdot}(P{\cdot}x{\cdot}y) & = & y \quad . \end{array}$$

□

We call this a *specification* because it is a formal rendering of the properties P, L, R are *required* to have: this is not a definition (yet). Notice that this does not specify exactly what $P{\cdot}x{\cdot}y$ is, but only what the application of either L or R yields: P is only specified *implictly*. In programming, implicit specifications are quite common, and in functional programming this is no different.

This specification contains two equations that are not admissible. How do we solve them? To obtain an admissible equation for L we must transform the above specification into one of the shape $L{\cdot}x =$ "some expression", so, one way or the other, we must eliminate the subexpression $P{\cdot}x{\cdot}y$ from the left-hand side of L's specification.

To achieve this we observe that we can make the right-hand sides of these equations more uniform, by means of the combinators K and C from the previous subsection (for all x, y):

$$\begin{array}{lcl} L{\cdot}(P{\cdot}x{\cdot}y) & = & K{\cdot}x{\cdot}y \quad , \\ R{\cdot}(P{\cdot}x{\cdot}y) & = & C{\cdot}x{\cdot}y \quad . \end{array}$$

The common shape of both right-hand sides now is $z{\cdot}x{\cdot}y$, where either K or C is substituted for z. So, z is a parameter of the pair as well, for which either K or C is supplied to select either the left or the right element. Therefore, we let $P{\cdot}x{\cdot}y$ be a function, declared as follows.

declaration:

$$P\cdot x\cdot y\cdot z \;=\; z\cdot x\cdot y \quad .$$

□

This is a *design decision*; inventing this declaration is the hard part of this problem. Once this decision has been taken, we can derive solutions for L and R quite easily:

$$\begin{array}{ll}
 & L\cdot(P\cdot x\cdot y) \;=\; x \\
\equiv & \quad \{ \text{ as above } \} \\
 & L\cdot(P\cdot x\cdot y) \;=\; K\cdot x\cdot y \\
\equiv & \quad \{ \text{ declaration of } P\text{, with } z{:=}K \ \} \\
 & L\cdot(P\cdot x\cdot y) \;=\; P\cdot x\cdot y\cdot K \\
\equiv & \quad \{ \text{ inserting parentheses, to clarify the structure } \} \\
 & L\cdot(P\cdot x\cdot y) \;=\; (P\cdot x\cdot y)\cdot K \\
\Leftarrow & \quad \{ \text{ abstraction, see below } \} \\
 & (\forall p :: \; L\cdot p = p\cdot K\,) \quad .
\end{array}$$

In a very similar way we can derive that $(\forall p :: \; R\cdot p = p\cdot C)$ satisfies the specification of R. Thus, we obtain a complete solution for P, L, R, in terms of combinators K, C defined earlier.

declaration:

$$\begin{array}{lrcl}
 & P\cdot x\cdot y\cdot z & = & z\cdot x\cdot y \\
\& & L\cdot p & = & p\cdot K \\
\& & R\cdot p & = & p\cdot C
\end{array}$$

□

In the derivation of a declaration for L we have taken a step labelled "abstraction"; the reasoning in such a step is: to satisfy an equality of the shape $L\cdot E \;=\; F[\,E\,]$, where E is an expression and $F[\,E\,]$ is an expression having E as a subexpression, we choose to let L satisfy the more general (and stronger) $(\forall x :: \; L\cdot x = F[\,x\,])$. Thus, we have replaced the specific subexpression E by the unspecific parameter x –whence the label abstraction–; the result, $L\cdot x \;=\; F[\,x\,]$, is an admissible equation satisfying the requirement.

In the reasoning leading to the declaration of P we have identified x and y as special cases, or: "instances", of a common pattern, namely the more

general expression $z \cdot x \cdot y$. This technique is called *generalisation*, of which we will see many more applications throughout this text.

1.2.2 booleans

The datatype *Boolean* consists of two constants *true* and *false* , and a set of boolean operations. It so happens that all boolean operations can be defined by means of a single primitive function, which we call *if* here. Function *if* is a (so-called) *selector*: it takes a boolean argument and two other arguments, and depending on the value of the boolean argument its value either is the second or the third argument. Thus, we obtain the following specification for *if* . Again, this specification is not a declaration, because *true* and *false* do not appear here as parameters, but as free variables for which declarations are still to be provided.

specification 7 : for all x, y :

$$\begin{array}{lcl} \mathit{if} \cdot \mathit{true} \cdot x \cdot y & = & x \quad , \\ \mathit{if} \cdot \mathit{false} \cdot x \cdot y & = & y \quad . \end{array}$$

□

The two boolean values *true* and *false* must be different, and one might well wonder whether we should not include this in the specification. Fortunately, though, this is not necessary because the above specification already implies that *true* and *false* are different:

$$\begin{array}{ll} & \mathit{true} \neq \mathit{false} \\ \Leftarrow & \quad \{ \text{ all we have is specification 7 : Leibniz } \} \\ & \mathit{if} \cdot \mathit{true} \neq \mathit{if} \cdot \mathit{false} \\ \Leftarrow & \quad \{ \text{ Leibniz (twice) } \} \\ & (\exists x, y :: \mathit{if} \cdot \mathit{true} \cdot x \cdot y \neq \mathit{if} \cdot \mathit{false} \cdot x \cdot y \,) \\ \equiv & \quad \{ \text{ specification 7 } \} \\ & (\exists x, y :: x \neq y \,) \\ \equiv & \quad \{ \text{ property 4 } \} \\ & \mathsf{true} \quad . \end{array}$$

By proving this result from the specification, instead of from a solution, we know that this result is independent of any solution; the advantage of this is

that we do not have to reprove the result whenever we would decide to replace the solution by a new one.

In terms of if, $true$, and $false$, all other boolean operations can be defined; for example:

declaration 8:

$$\begin{array}{rrcl} & non\cdot x & = & if\cdot x\cdot false\cdot true \\ \& & and\cdot x\cdot y & = & if\cdot x\cdot y\cdot false \\ \& & or\cdot x\cdot y & = & if\cdot x\cdot true\cdot y \end{array}$$

□

Hence, it suffices to construct declarations for if, $true$, and $false$ only. One possible solution can be obtained as follows:

$$\begin{array}{ll} & if\cdot true\cdot x\cdot y \;=\; x \\ \equiv & \quad \{ \text{ declaration of } K \text{, to introduce } y \text{ on the right-hand side } \} \\ & if\cdot true\cdot x\cdot y \;=\; K\cdot x\cdot y \\ \Leftarrow & \quad \{ \text{ Leibniz (twice) } \} \\ & if\cdot true \;=\; K \\ \equiv & \quad \{ \text{ design decision: choose } if = I \} \\ & I\cdot true \;=\; K \\ \equiv & \quad \{ \text{ declaration of } I \} \\ & true \;=\; K \quad . \end{array}$$

So, if we choose $if = I$ then $true = K$ satisfies the first line of the specification; in very much the same way we obtain, with the same choice for if, that $false = C$ satisfies the second line of the specification. This way, we obtain a set of declarations meeting all requirements.

declaration:

$$\begin{array}{rrcl} & if & = & I \\ \& & true & = & K \\ \& & false & = & C \end{array}$$

□

1.2.3 natural numbers

According to Peano, the datatype *Natural* of natural numbers is the set of values defined by means of a constant *zero* and a function *succ*, in such a way that natural number n is represented in *Natural* by $succ^n \cdot zero$.

specification 9 (part i): for all $x \in Natural$:

$$zero \in Natural \quad , \text{ and}$$
$$succ \cdot x \in Natural \quad .$$

□

Type *Natural* is the *smallest* of all sets satisfying this specification, but to obtain a useful datatype we need more. First, we need a possibility to distinguish *zero* from values of the shape $succ \cdot x$; second, we need a function to obtain x back from $succ \cdot x$, that is, we need the inverse of *succ*. For these purposes, we introduce two additional functions, *iszero* and *pred*.

specification 9 (part ii): for all $x \in Natural$:

$$\begin{aligned} iszero \cdot zero &= true \quad , \\ iszero \cdot (succ \cdot x) &= false \quad , \text{ and} \\ pred \cdot (succ \cdot x) &= x \quad . \end{aligned}$$

□

This part of the specification guarantees that all values "generated" by means of *zero* and *succ* really are different; this is necessary because different natural numbers must be represented differently.

property 10: for all natural m, n:

$$m \neq n \quad \Rightarrow \quad succ^m \cdot zero \neq succ^n \cdot zero \quad .$$

proof: (exercise 7)
□

Specification 9 admits several solutions. Here is a solution in terms of the pair-forming combinators P, L, and R, and the booleans. This solution is inspired by the observation that pairing with the boolean constants is one way to guarantee distinguishability: the booleans are used as *tags* here. The proof that the following declaration indeed implies the above specification is left to exercise 7.

declaration 11 :

$$\begin{array}{rrcl} & zero & = & P\cdot true\cdot y \ \ \mathsf{whr}\ y=y\ \ \mathsf{end} \\ \& & succ\cdot x & = & P\cdot false\cdot x \\ \& & iszero\cdot x & = & L\cdot x \\ \& & pred\cdot x & = & R\cdot x \end{array}$$

□

1.2.4 a black hole

A rather strange but perfectly admissible object is function B declared recursively by:

$$B\cdot x \ = \ B \ .$$

B is a function that, when applied to an argument, yields itself again; the argument is just swallowed, so to speak, without leaving any trace. This can be repeated indefinitely: B can applied to as many arguments as you like, and still yield nothing but itself.

Within our universe Ω function B acts as a "black hole", although traditionally B is called "bottom" –despite its bottomless insatiability– , denoted by the symbol $\perp$. It can be viewed as the ultimately undefined value, or as the computation that continues indefinitely, consuming input without producing any output.

1.2.5 exercises

In these exercises variables I, K, C, P, and B are the ones declared in the previous examples.

0. Prove that function composition is associative, by showing, for all x, that $((f\circ g)\circ h)\cdot x$ equals $(f\circ(g\circ h))\cdot x$.

1. Prove that $f\circ g$ has type $X\to Z$ whenever g has type $X\to Y$ and f has type $Y\to Z$.

2. Using only specification 7 for $if, true, false$, prove that non, and, or –see declaration 8 – have their "usual properties".

3. Choose declarations for functions imp and eqv, such that they represent boolean implication and equivalence, respectively. Prove the correctness of your solution.

4. Prove:
   a. $if \cdot B \cdot x \cdot y \;=\; B$, for all x, y
   b. $non \cdot B \;=\; B$
   c. $and \cdot B \cdot y \;=\; B$
   d. $and \cdot false \cdot B \;=\; false$

5. Prove:
   a. $true \neq B$
   b. $P \cdot B \cdot B \neq B$
   c. $K \neq C$
   d. $succ \cdot x \neq B$, for all x

6. Suppose function tw – "twice" – is declared by:

$$tw \cdot x \;=\; x \circ x \quad .$$

   What is the value of the expression $tw \cdot tw \cdot tw \cdot succ \cdot zero$?

7. Prove that the specification of type $Natural$ guarantees that its elements are different, that is, prove property 10 using specification 9 only.

8. Prove that declaration 11 satisfies specification 9 .

9. a. Formulate a specification of a function add, having type $Natural \rightarrow (Natural \rightarrow Natural)$, such that add represents addition of natural numbers.
   b. Construct a declaration for add, and show that it satifies its specification.
   c. Construct a specification and a declaration for a function $subt$ that represents subtraction.

10. What is, in declaration 11 , the role of declared variable y in the expression $P \cdot true \cdot y$ whr $y = y$ end ? Is it possible to avoid its presence?

11. The simplest possible expression containing both x and y is $x \cdot y$; in view of this we might, as solution for the pair-forming operator P , propose $P \cdot x \cdot y \;=\; x \cdot y$. Prove that this proposal fails, by showing that no function l exists satisfying: $(\forall x, y :: l \cdot (x \cdot y) = x)$.

12. *a fixed point operator*: Derive, construct, or just invent, a *non-recursive* declaration of a function *fix* satisfying: $(\forall x :: \mathit{fix}\cdot x = x\cdot(\mathit{fix}\cdot x))$.

1.3 Epilogue

The applicative algebra is a variation of the λ-calculus, in a form more suitable for programming. In particular, the where-clauses of the applicative algebra correspond to the lambda abstractions of the λ-calculus in a simple way; for non-recursive declarations $f\cdot x\cdot y = E$ we have:

$$(f \ \mathsf{whr}\ f\cdot x\cdot y = E \ \mathsf{end}) \ = \ (\lambda x, y : E) \quad .$$

The Rule of Instantiation in the applicative algebra is the direct counterpart of the rule for β-conversion in the λ-calculus; in the applicative algebra we have:

$$(f \ \mathsf{whr}\ f\cdot x\cdot y = E \ \mathsf{end})\cdot A\cdot B \ = \ E(x := A)(y := B) \quad ,$$

whereas in the λ-calculus we have:

$$(\lambda x, y : E)\cdot A\cdot B \ = \ E(x := A)(y := B) \quad .$$

Recursive declarations can be transformed into equivalent non-recursive ones by means of a (so-called) *fixed-point* operator *fix* –see exercise 11– , satisfying:

$$(\forall x :: \mathit{fix}\cdot x = x\cdot(\mathit{fix}\cdot x)) \quad ;$$

for example, the possibly recursive declaration of the shape:

$$f\cdot x\cdot y \ = \ E \quad ,$$

can be replaced by the (functionally) equivalent but non-recursive declaration:

$$f \ = \ (\mathit{fix}\cdot g \ \ \mathsf{whr}\ g\cdot f\cdot x\cdot y \ = \ E \ \mathsf{end}) \quad .$$

In this declaration the recursive occurrences, in expression E , of variable f have been bound: in E , function f now is a parameter of the auxiliary function g .

* * *

In Section 1.1 we have argued that the elements of Ω, and the solutions to admissible equations in particular, are not functions but that they can be used as functions, of type $\Omega \to \Omega$. One of the reasons for this attitude is to avoid problems with expressions like $f \cdot f$, which are perfectly valid in the applicative algebra: functions usually cannot be applied to themselves, so such expressions would be meaningless. By playing the game in our way we avoid these problems: in $f \cdot f$ a function is applied, not to itself but to a representation of itself; as a result, self-application is a non-problem.

* * *

That the whole of the applicative algebra can be characterised by just two axioms is quite nice. Both these axioms are needed, and they complement each other. On the one hand, the Dot Axiom is needed to guarantee the existence of solutions to certain equations, but it only yields equalities. The Anti-Singleton Axiom, on the other hand, is needed to preclude the useless case that all expressions in the language would be equal: this axiom yields the necessary differences.

Actually, these two axioms even suffice to justify the conclusion that Ω is *infinite*. This requires no separate proof anymore, because we have already shown –see property 10– that the natural numbers are representable in Ω.

1997.11.1 2013.4.15 (correction of a few simple errors)

Chapter 2

Simple Extensions

2.0 Introduction

The applicative algebra is the core of the functional programming language. In this chapter we extend the applicative algebra with a few basic data-types and with notations for (guarded) selections and for tuple formation. In a later chapter we will introduce a notation for list structures.

Very deliberately, we have kept the programming language as sober as possible: this text is about programming techniques, not about (however useful) language features. We only introduce those concepts that are necessary or convenient to develop and illustrate a style of programming. In particular, we do not care to think about such practical matters as, for example, *characters*, *strings*, or *floating point numbers*, nor will we introduce much syntactic sugar for *data types* –abstract, concrete, algebraic, recursive, generic, or what have you– or *type classes*, as can be found in modern functional programming languages: for our purposes, tuples and lists will be sufficient.

2.1 Basic Datatypes

The basic data types are the sets Bool, Nat, and Int, defined informally by:

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

Notice that, according to this informal definition, Nat is a true subtype of Int. As these sets and, the operators defined on them, are well-known, we do

not formulate more formal definitions here. We just give a summary of the operators we will be using throughout this text.

summary 0: Together with their types, the arithmetic, relational, and boolean operators are, in order of decreasing binding power:

operator	type
(unary) $-$	$\mathsf{Int} \to \mathsf{Int}$
$*$, div , mod	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
$+$, $-$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
max , min	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
$<$, $\leq$, $=$, $\neq$, $\geq$, $>$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$
(unary) $\neg$	$\mathsf{Bool} \to \mathsf{Bool}$
$\wedge$, $\vee$	$\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$
$\Leftarrow$, $\Rightarrow$	$\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$
$\equiv$, $\not\equiv$	$\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$

Some operators, like $*, +$, max, min , also have type $\mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ (cf. definition 1.6). We will use div and mod with positive divisors only, in which case the pair $(b \,\mathsf{div}\, c, b \,\mathsf{mod}\, c)$ is the unique pair (q, r) of integers satisfying:

$$b = q * c + r \;\wedge\; 0 \leq r < c \quad .$$

□

Expressions having type Bool , Nat , and Int are called *boolean*, *natural*, and *integer* expressions respectively. From Chapter 1 we recall that "having a type" is not a syntactical property of expressions, but one that, like any other property, requires proof. Such proofs involve, of course, the above typing rules for the operators.

simple example:

$$\begin{array}{ll} & 0 \leq x - 1 \;\in\; \mathsf{Bool} \\ \Leftarrow & \quad \{ \text{ type of } \leq \text{ } \} \\ & 0 \in \mathsf{Int} \;\wedge\; x - 1 \in \mathsf{Int} \\ \Leftarrow & \quad \{ \text{ type of } - \text{ } \} \end{array}$$

$$
\begin{array}{ll}
 & 0 \in \mathsf{Int} \ \wedge \ x \in \mathsf{Int} \ \wedge \ 1 \in \mathsf{Int} \\
\equiv & \quad \{ \text{ definition of } \mathsf{Int} \ \} \\
 & x \in \mathsf{Int} \ ,
\end{array}
$$

where the validity of $x \in \mathsf{Int}$ depends on the type given to or assumed about variable x.

□

2.2 Operator Sections

For any binary operator $\oplus$, and for any two constants b, c (of the right type), we sometimes will have need of functions f, g, or h, defined by:

$$
\begin{array}{rcl}
f \cdot y & = & b \oplus y \ , \\
g \cdot x & = & x \oplus c \ , \\
h \cdot x \cdot y & = & x \oplus y \ .
\end{array}
$$

To avoid the need to choose names for these functions, we will use a shorthand notation and write $(b\oplus)$ for f, and $(\oplus c)$ for g, and $(\oplus)$ for h; hence, for all x, y we have:

$$
\begin{array}{rcl}
(b\oplus) \cdot y & = & b \oplus y \ , \\
(\oplus c) \cdot x & = & x \oplus c \ , \\
(\oplus) \cdot x \cdot y & = & x \oplus y \ .
\end{array}
$$

examples: $(+1)$ is the integer successor function, $(\cdot\, c)$ –"apply to c"– is the function that applies its (function) argument to c, whereas $(f \cdot)$ is just f; furthermore, (<0) is the boolean function that is true on the negative integers and false otherwise, and the predicate characterizing the even integers can be compactly written as $(=0) \circ (\mathsf{mod}\, 2)$. Notice that expressions like (-3) are ambiguous now, but in practice this hardly causes confusion.

□

2.3 Guarded Selections

In the process of constructing an expression satisfying a given specification R, it sometimes happens that we can only come up with expressions that satisfy R *conditionally*, that is, provided the variables involved satisfy additional

conditions. Formally, this means that, instead of having an expression E satisfying $R{\cdot}E$, we have obtained an expression E satisfying:

$$B \Rightarrow R{\cdot}E \quad ,$$

where B is a boolean expression representing the additional condition[0]. Such an expression is a partial solution to the problem: only for those values of the variables that satisfy B, expression E satisfies $R{\cdot}E$. We call such a boolean condition a *guard*, and we write expression E together with its guard B as a so-called *guarded expression*, thus:

$$B \rightarrow E \quad .$$

A *guarded selection* provides a way to combine several such guarded expressions into one single expression. For this we use a notation in which –in contrast to other functional languages– the textual order of the guarded expressions is irrelevant for their meaning. As a result, each guarded expression can be completely understood in isolation, that is, without the context of its surrounding guarded expressions[1].

A guarded selection now is an expression composed from one or more *guarded expressions*, also called *alternatives*, of the shape $B_i \rightarrow E_i$, for all i: $0 \leq i \leq n$; here B_i is a boolean expression and E_i is an expression, for each i. The alternatives are connected by a bar symbol $[]$, and bracketed by a pair $\mathsf{if} \cdots \mathsf{fi}$. So, the general shape of a guarded selection with $n{+}1$ alternatives is:

$$\mathsf{if} \;\; B_0 \rightarrow E_0 \;\; [] \;\; \cdots \;\; [] \;\; B_n \rightarrow E_n \;\; \mathsf{fi} \quad .$$

A guarded selection satisfies a given specification if, first, each guarded expression provides a partial solution to that specification, as explained above, and, second, if the guards together cover all possible cases. This is reflected by the following rule, in which the requirement that the guards be boolean has been made explicit as well.

rule 1: For predicate R and a guarded selection GS composed from $n{+}1$ alternatives $B_i \rightarrow E_i$, for $0 \leq i \leq n$, we have that:

$$R{\cdot}\,GS$$

follows from the conjunction of these three conditions:

[0]Eigenlijk moet ik $[\; B \Rightarrow R{\cdot}E \;]$ schrijven, maar ik heb weinig zin de rechte haken te introduceren. Is dat (eigen)wijs?

[1]Moet ik de hier gelijkenis met de guarded commands vermelden?

$$(\forall i :: B_i \in \mathsf{Bool}) \quad ,$$
$$(\exists i :: B_i) \quad ,$$
$$(\forall i :: B_i \Rightarrow R \cdot E_i) \quad .$$

□

example: With *E23* for the guarded selection:

$$\mathsf{if}\ \mathsf{true} \rightarrow 2\ [\!]\ \mathsf{true} \rightarrow 3\ \mathsf{fi} \quad ,$$

the conditions to justify the conclusion $R \cdot E23$ boil down to:

$$R \cdot 2 \ \wedge\ R \cdot 3 \quad ,$$

for every predicate R; the *strongest* R satisfying these conditions is defined by:

$$R \cdot x \ \equiv\ x = 2 \ \vee\ x = 3 \quad .$$

So, we may conclude that:

$$E23 = 2 \ \vee\ E23 = 3 \quad ,$$

and this is the best we can prove, by means of this rule, about expression $E23$. In particular, it is impossible to prove $E23 = 2$ in isolation.

□

To use the rule to prove that the value of a guarded selection equals a fixed value X, we must prove that each of the guarded expressions equals X; this is reflected by the following special case of the rule, which has been obtained by substitution of $(=X)$ for predicate R.

rule 2: For value X and a guarded selection GS composed from $n+1$ alternatives $B_i \rightarrow E_i$, for $0 \leq i \leq n$, we have that:

$$GS = X \quad ,$$

follows from the conjunction of these three conditions:

$$(\forall i :: B_i \in \mathsf{Bool}) \quad ,$$
$$(\exists i :: B_i) \quad ,$$
$$(\forall i :: B_i \Rightarrow E_i = X) \quad .$$

□

example: A valid (equivalent) expression for $x \,\mathsf{max}\, y$ is y, but only so if $x \leq y$; by prefixing expression y with this condition, we obtain

$x \leq y \rightarrow y$ as a guarded expression for $x \,\mathsf{max}\, y$. Similarly, we obtain $y \leq x \rightarrow x$ as another guarded expression for $x \,\mathsf{max}\, y$. Because either $x \leq y$ or $y \leq x$, these two guarded expressions cover all cases; by combining them into a single guarded selection we obtain:

$$\begin{array}{lcl} x \,\mathsf{max}\, y & = & \mathsf{if}\ \ y \leq x \ \rightarrow\ x \\ & & [\!]\ \ x \leq y \ \rightarrow\ y \\ & & \mathsf{fi} \end{array}$$

□

2.4 Tuples

2.4.0 notation

Two values can be combined into a single entity, called a *pair*, with the two values as its elements. Similarly, three values can be combined into a *triple*, and, generally, n values, for any natural n, can be combined into a, so-called, n*-tuple*.

From n expressions E_i , $0 \leq i < n$, we form an n-tuple by connecting the expressions by commas and by enclosing them by angular brackets $\langle \cdots \rangle$:

$$\langle E_0, \cdots, E_{n-1} \rangle \quad .$$

For $n = 0$ this yields the (one-and-only) *empty tuple*:

$$\langle \, \rangle \quad ,$$

whereas for $n = 1$ we obtain the *singletons*, of the shape:

$$\langle E_0 \rangle \quad ,$$

for $n = 2$ we obtain the *pairs*, of the shape:

$$\langle E_0, E_1 \rangle \quad ,$$

and so on.

2.4.1 element selection

The values making up a tuple are called its *elements*, so an n-tuple has n elements. To distinguish these elements we use the natural numbers $i : 0 \leq i < n$ as indices; mathematically, this means that an n-tuple is a function on the domain $i : 0 \leq i < n$. So, we have:

$$\langle E_0, \cdots, E_{n-1} \rangle \cdot i \ = \ E_i \quad , \ \text{for all } i : 0 \leq i < n \quad .$$

2.4.2 the size operator

Occasionally, we wish to define a function taking tuples of different sizes as its arguments; in the declaration of such a function we then need a way to distinguish tuples according to their sizes. For this purpose, we use the prefix-operator $\#$ ("size") that, when applied to a tuple, yields its size:

$$\#\langle E_0, \cdots, E_{n-1} \rangle \;=\; n \quad .$$

examples: The empty tuple has size 0, the singletons have size 1, and so on:

$$\begin{array}{rcl} \#\langle\,\rangle & = & 0 \\ \#\langle x \rangle & = & 1 \\ \#\langle x, y \rangle & = & 2 \\ \#\langle x, y, z \rangle & = & 3 \\ & \vdots & \end{array}$$

□

2.4.3 tuple types

We will write:

$$\langle X_0, \cdots, X_{n-1} \rangle \quad ,$$

for the type of all n-tuples formed from elements having types X_i , $0 \leq i < n$.

examples: $\langle x, y, z \rangle$ is a triple of type $\langle X, Y, Z \rangle$ whenever x, y, z have types X, Y, Z respectively. Also, the empty tuple $\langle\,\rangle$ has type $\langle\,\rangle$ –notice the (harmless) ambiguity here, such that $\langle\,\rangle \in \langle\,\rangle$ –. To resolve this ambiguity, we adopt the rule that, by default, $\langle\,\rangle$ denotes the empty tuple, and that we always will speak of "type $\langle\,\rangle$" to denote the type containing the empty tuple.

□

example: Operator $\#$ has types:

$$\begin{array}{rcl} \langle\,\rangle & \rightarrow & \mathsf{Nat} \\ \langle X \rangle & \rightarrow & \mathsf{Nat} \\ \langle X, Y \rangle & \rightarrow & \mathsf{Nat} \\ \langle X, Y, Z \rangle & \rightarrow & \mathsf{Nat} \\ & \vdots & \end{array}$$

□

2.5 Declaration patterns

Guarded selections offer the possibility to combine several guarded expressions into one, single expression. In calculational derivations of function declarations, however, guarded selections often are too large and, therefore, awkward to manipulate and carry around: usually, we prefer to construct separate derivations for each of the alternatives in isolation.

By means of so-called *declaration patterns* the structure of the more common forms of case analysis can be reflected in the way we write function declarations. Declaration patterns are just abbreviations of declarations containing guarded selections, in such a way that the alternatives are written in a more isolated form, matching the forms used in derivations more closely.

2.5.0 isolated alternatives

A single declaration of a function f, whose defining expression is a guarded selection with $n+1$ alternatives, like:

$$\begin{array}{lcl} f\cdot x & = & \mathsf{if}\ B_0 \rightarrow E_0 \\ & & [\!]\ B_1 \rightarrow E_1 \\ & & \vdots \qquad \vdots \\ & & [\!]\ B_n \rightarrow E_n \\ & & \mathsf{fi} \quad , \end{array}$$

may also be written as a bunch of $n+1$ separate declarations for *one-and-the-same* function f:

$$\begin{array}{lcl} & f\cdot x & = & \mathsf{if}\ B_0 \rightarrow E_0\ \mathsf{fi} \\ \& & f\cdot x & = & \mathsf{if}\ B_1 \rightarrow E_1\ \mathsf{fi} \\ \vdots & & \vdots & \\ \& & f\cdot x & = & \mathsf{if}\ B_n \rightarrow E_n\ \mathsf{fi} \quad . \end{array}$$

The advantages of this form are that each of the alternatives in such a bunch can now be discussed in isolation and that each of them can be abbreviated into a form matching our manipulative needs.

2.5.1 patterns for the naturals

A declaration of the shape, where E is an expression:

$$f\cdot 0 \ = \ E \quad ,$$

is an abbreviation of the declaration –in which variable x is fresh–:

$$f\cdot x \;=\; \mathsf{if}\ x{=}0 \rightarrow E\ \mathsf{fi}\ \ .$$

Similarly, a declaration of the shape:

$$f\cdot(n{+}1) \;=\; E\ \ ,$$

is an abbreviation of the declaration –in which variable x is fresh again–:

$$f\cdot x \;=\; \mathsf{if}\ x{\geq}1 \rightarrow E\ \mathsf{whr}\ n{=}x{-}1\ \mathsf{end}\ \mathsf{fi}\ \ .$$

example: This declaration:

$$\begin{array}{lrcl} & \mathit{fib}\cdot 0 & = & 0 \\ \& & \mathit{fib}\cdot 1 & = & 1 \\ \& & \mathit{fib}\cdot(n{+}2) & = & \mathit{fib}\cdot n + \mathit{fib}\cdot(n{+}1) \end{array}$$

is equivalent with this one:

$$\begin{array}{rcl} \mathit{fib}\cdot x & = & \mathsf{if}\ \ x{=}0 \rightarrow\ 0 \\ & & []\ \ x{=}1 \rightarrow\ 1 \\ & & []\ \ x{\geq}2 \rightarrow\ \mathit{fib}\cdot n + \mathit{fib}\cdot(n{+}1)\ \mathsf{whr}\ n{=}x{-}2\ \mathsf{end} \\ & & \mathsf{fi} \end{array}$$

□

2.5.2 patterns for tuples

Occasionally, we wish to introduce (local) names for the elements of a tuple without need of a name for the tuple itself. For this purpose we will use a declaration of the shape:

$$\langle b, c, d\rangle \;=\; E\ \ ,$$

as an abbreviation of the multiple declaration:

$$b \;=\; E\cdot 0\ \ \&\ \ c \;=\; E\cdot 1\ \ \&\ \ d \;=\; E\cdot 2\ \ .$$

A declaration of the shape, where E is an expression:

$$f\cdot\langle\ \rangle \;=\; E\ \ ,$$

is an abbreviation of the declaration –in which variable x is fresh–:

$$f\cdot x \;=\; \mathsf{if}\ \#x{=}0 \rightarrow E\ \mathsf{fi}\ \ .$$

Similarly, a declaration of the shape:

$$f\cdot\langle b, c, d\rangle \;=\; E\ \ ,$$

is an abbreviation of the declaration –in which variable x is fresh again–:

$$f\cdot x \;=\; \mathsf{if}\ \#x{=}3 \rightarrow E\ \mathsf{whr}\ \langle b, c, d\rangle{=}x\ \mathsf{end}\ \mathsf{fi}\ \ .$$

2.6 Examples and Exercises

2.6.0 currying and uncurrying

With the availability of tuples a function of many parameters can now be declared in two ways: either as a function with as many parameters as needed or as a function with one parameter that is a tuple with as many elements as needed. For example, for the same expression E we can declare functions f and g mapping x and y to E in two ways. If E has type V, for all x of type X and y of type Y, then f has type $X \to (Y \to V)$ whereas g has type $\langle X, Y \rangle \to V$:

$$\begin{array}{lll} & f \cdot x \cdot y & = & E \\ \& & g \cdot \langle x, y \rangle & = & E \end{array}$$

Now we have $f \cdot x \cdot y = g \cdot \langle x, y \rangle$. The definitions of f and g can even be transformed into one another, by means of two functions *curry* and *uncurry*, declared thus:

$$\begin{array}{lll} & \mathit{curry} \cdot g \cdot x \cdot y & = & g \cdot \langle x, y \rangle \\ \& & \mathit{uncurry} \cdot f \cdot \langle x, y \rangle & = & f \cdot x \cdot y \end{array}$$

Thus, we have $f = \mathit{curry} \cdot g$ and $g = \mathit{uncurry} \cdot f$. These conversions are useful because sometimes it is more convenient to treat a many-parameter function as a function of one single parameter, or the other way around.

2.6.1 tupled functions

Tuples play an important role in *program transformations*, which are directed towards improvement of a program's efficiency. Here is a simple example giving the flavour of this game. Tupling as a transformation technique will be discussed more extensively later.

The well-known *Fibonacci sequence* is a function *fib* on the naturals, declared as follows:

$$\begin{array}{lrcl} & \mathit{fib} \cdot 0 & = & 0 \\ \& & \mathit{fib} \cdot 1 & = & 1 \\ \& & \mathit{fib} \cdot (n{+}2) & = & \mathit{fib} \cdot n + \mathit{fib} \cdot (n{+}1) \end{array}$$

We now introduce two new functions f and g which we require to satisfy –so this is a specification, not a definition–:

$$(\forall n : 0 \leq n : f \cdot n = \mathit{fib} \cdot n \ \wedge \ g \cdot n = \mathit{fib} \cdot (n{+}1)) \quad .$$

From this and the above declaration of *fib* we can derive recursive declarations for f and g, yielding:

$$\begin{array}{lrcl} & f\cdot 0 & = & 0 \\ \& & g\cdot 0 & = & 1 \\ \& & f\cdot(n{+}1) & = & g\cdot n \\ \& & g\cdot(n{+}1) & = & f\cdot n + g\cdot n \end{array}$$

Because *fib* does not occur in these declarations anymore we can now use them to define *fib*, simply by declaring *fib* to be equal to f.

The declarations of f and g exhibit the same pattern of recursion: both $f\cdot 0$ and $g\cdot 0$ are declared directly, and both $f\cdot(n{+}1)$ and $g\cdot(n{+}1)$ are declared recursively in terms of $f\cdot n$ and $g\cdot n$. This makes it possible to introduce a function h, of type $\mathsf{Nat}\to\langle\mathsf{Nat},\mathsf{Nat}\rangle$, such that $h\cdot n$ is the pair of values $f\cdot n$ and $g\cdot n$; in the jargon we say that h is obtained by pairing f and g:

$$(\forall n : 0\le n : h\cdot n = \langle f\cdot n, g\cdot n\rangle) \quad .$$

A declaration for h satisfying this specification is:

$$\begin{array}{lrcl} & h\cdot 0 & = & \langle 0,1\rangle \\ \& & h\cdot(n{+}1) & = & \langle y, x{+}y\rangle \ \mathsf{whr}\ \langle x,y\rangle = h\cdot n \ \mathsf{end} \end{array}$$

2.6.2 recursive datatypes

The datatype of binary trees (all) whose *nodes* are labelled with integers can be defined as the smallest subset T of the value domain Ω satisfying:

$$\begin{array}{rl} \langle\ \rangle \in T & \text{, and} \\ \langle b,s,t\rangle \in T & \text{, for all } b\in\mathsf{Int} \text{ and } s,t\in T \quad . \end{array}$$

Similarly, the datatype of binary trees of which (only) the *leaves* are labelled with integers can be defined as the smallest subset T of Ω satisfying:

$$\begin{array}{rl} \langle b\rangle \in T & \text{, for all } b\in\mathsf{Int} \text{ , and} \\ \langle s,t\rangle \in T & \text{, for all } s,t\in T \quad . \end{array}$$

2.6.3 exercises

0. For integer b and positive c, prove $b\,\mathsf{div}\,c\ge 0 \equiv b\ge 0$, using the definition of div given in summary 0.

1. Prove $x=y \equiv (\forall z : z\in\mathsf{Int} : x\le z \equiv y\le z)$, for all integer x,y.

2. Prove $x \,\mathsf{max}\, y \leq z \;\equiv\; x \leq z \wedge y \leq z$, for all integer x, y, z .

3. Prove that $+$ distributes over max .

4. Prove: a function f of type $\mathsf{Int} \to \mathsf{Int}$ is monotonic if and only if f distributes over max .

5. Construct an expression for the absolute value of integer x .

6. Eliminate the declaration patterns from the following declaration, by rewriting it into full, unabbreviated form:

$$\begin{array}{lll} & v\cdot\langle b\rangle & = & b \\ \& & v\cdot\langle 0,s,t\rangle & = & v\cdot s \,+\, v\cdot t \\ \& & v\cdot\langle 1,s,t\rangle & = & v\cdot s \,*\, v\cdot t \end{array}$$

7. With functions fib, f, g, h as declared in Section 2.6.1, prove: $(\forall n : 0 \leq n : f\cdot n = \mathit{fib}\cdot n)$; also prove $(\forall n : 0 \leq n : h\cdot n = \langle f\cdot n, g\cdot n\rangle)$.

8. Construct a declaration for function f , of type: $\langle \mathsf{Int}, \mathsf{Int}, \mathsf{Int}\rangle \to \{0,1,2\}$, satisfying:

$$f\cdot\langle b,c,d\rangle \;=\; \text{“the number of real solutions of the quadratic equation: } x : b*x^2+c*x+d=0\text{”} \quad .$$

9. To obtain a solution of the quadratic equation $x : b*x^2+c*x+d=0$, a functional programmer has proposed this (recursive) declaration:

$$x \;=\; -(b*x^2+d)/c \quad .$$

Explain why this is no good. (hint: see Section 2.7.)

2.7 Epilogue

With the integers part of the language, this is a valid declaration:

$$x \;=\; 1+x \quad ,$$

and the corresponding admissible equation, $x : x = 1+x$, has a solution in the value domain Ω . We also know that this equation has no integer solutions, so we have:

$$(\exists x : x \in \Omega : \ x = 1 + x) \quad , \quad \text{but also}$$
$$\neg\,(\exists x : x \in \mathsf{Int} : \ x = 1 + x) \quad .$$

This shows that, in whatever way the integers are embedded in Ω, they will never cover the whole of it: Ω always will contain "strange" values, like solutions of the equation $x : \ x = 1 + x$. This has far-reaching consequences. For example, this is an admissible equation too:

$$x : \ x \ = \ 2 - x \quad ,$$

of which we might be tempted to believe that it has 1 as its unique solution. Within the integers, its solution is unique indeed, but within Ω we cannot preclude that the solutions of $x : \ x = 1 + x$ are solutions of $x : \ x = 2 - x$ as well. Therefore, equations like these are rather useless.

These pathological examples are a direct consequence of the underlying computational model, where declarations are treated as left-to-right rewrite rules: with, for example, x declared by $x = 2 - x$, evaluation of expression x will give rise to a nonterminating computation:

$$\begin{array}{ll}
 & x \\
= & \quad \{ \text{ declaration of } x \ \} \\
 & 2 - x \\
= & \quad \{ \text{ declaration of } x \ \} \\
 & 2 - (2 - x) \\
= & \quad \{ \text{ declaration of } x \ \} \\
 & 2 - (2 - (2 - x)) \\
= & \quad \{ \text{ declaration of } x \ \} \\
 & 2 - (2 - (2 - (2 - x))) \quad , \\
 & \text{and so on, indefinitely.}
\end{array}$$

This computation will not terminate and will not yield the integer solution we may have been hoping for; the evaluation of integer expressions, on the other hand, always terminates. This is another way of showing that Ω contains "strange" –non-integer– elements. In Chapter 5 we will discuss this computational model in more detail.

1998.3.10 2000.8.28 (minor revisions)

Chapter 3

Elementary Programming Techniques

3.0 Introduction

In the previous chapters we have introduced a simple notation for functional programs; together with the predicate calculus – needed to reason about programs – this notation provides the tool to construct programs in a calculational way. Now we are ready to start deriving programs.

In this chapter we will just play the game, without paying very much attention to its rules: the emphasis is on elementary *techniques* for the construction of solutions. In a later chapter we will elaborate on the rules and the nature of the game in greater detail.

The style of presentation is very tutorial here, and deliberately so. This is not meant as pedantry: experience shows that, both for beginning and for more advanced programmers, a well-developed and explicit awareness of the techniques and of how they are used contributes significantly to their effectiveness. Moreover, this gives the reader the opportunity to get used to this style, which at first may appear somewhat unconventional.

* * *

Different programs for the same problem may differ in their (running time) *efficiency*. To illustrate this we will state the efficiency of each program, without further explanation: the techniques needed to analyse a program's efficiency will be introduced later. Efficiency considerations are a main driving force behind program transformations.

3.1 Generalizations

An important problem solving technique is *generalization*. Every experienced programmer or mathematician uses it, maybe even without being aware. For example, ask a mathematician how to compute the 137-th prime number and he will immediately start thinking about the more general problem how to compute the n-th prime number, for any natural n: our mathematician knows that the particular value 137 is irrelevant and that the real problem is how to compute prime numbers. In this setting "irrelevant" means that computing the 137-th prime number probably is neither easier nor more difficult than computing the 136-th or the 138-th (or any other) prime number.

Most problems admit many generalizations, so it is important to have a way to choose the right ones. Here we intend to show how useful generalizations can be found by analysing what two (or more) similar expressions have in common: if these expressions can be viewed as instances of a single, more general expression, then this latter one may be the generalization we are looking for. Put differently, it is easier to find meaningful generalizations by looking at the differences between two (or more) similar expressions, rather than just considering a single one.

simple examples: What do 0 and 1 have in common? That they are natural numbers. Of course, they also have in common that they are the members of the set $\{0,1\}$, but in particular when the 1 emerges in the discussion as the successor of the 0, we are likely to encounter the need for the successor of x for any x under consideration. The smallest set containing both 0 and the successors of all its elements is, of course, the natural numbers.

Similarly, what do x and $x+1$ have in common? Well, because $+$ has 0 as its identity element, we have $x = x+0$; now, $x+0$ and $x+1$ are instances of the more general expression $x+y$, for any natural y.

□

3.1.0 from constants to functions

We start with the following example. For some fixed natural number N we are interested in the *smallest* natural number x satisfying $N \leq 2^x$.

Before we can solve this problem in a systematic way we must formalize it. Such a formal rendering of a problem is called a *specification*. We call a formula a specification to emphasize that it is not a proven fact but a problem to be solved: a specification (yet) to be satisfied has the same status as a

theorem to be proved or as an equation to be solved. One and the same problem usually admits several different specifications; therefore, which one we choose is important.

Here is a suitable specification for our current problem.

(0) $\quad x: \;\; 0 \le x \;\wedge\; N \le 2^x \;\wedge\; (\forall i : 0 \le i < x : 2^i < N) \;\; .$

The first two conjuncts express that x is a natural satisfying $N \le 2^x$, whereas the last conjunct expresses that x is the smallest such number. The prefix "$x:$" signals that we are looking for an x, not for an N (or anything else). When no confusion is possible we often omit such prefix, but strictly speaking it must always be there.

Because universal quantification over an empty range is true, in our case if $x=0$, it pays to investigate whether 0 solves the problem. It does, of course, if (and only if) $N \le 2^0$. For the complementary case $2^0 < N$ we now calculate:

$$
\begin{array}{ll}
 & 0 \le x \;\wedge\; N \le 2^x \;\wedge\; (\forall i : 0 \le i < x : 2^i < N) \\
\equiv & \quad \{\; 2^0 < N, \text{ hence we have: } N \le 2^x \Rightarrow 0 \ne x \;\} \\
 & 1 \le x \;\wedge\; N \le 2^x \;\wedge\; (\forall i : 0 \le i < x : 2^i < N) \\
\equiv & \quad \{\; \text{range split: } i=0 \vee 1 \le i < x \text{, permitted in view of } 1 \le x \;\} \\
 & 1 \le x \;\wedge\; N \le 2^x \;\wedge\; 2^0 < N \;\wedge\; (\forall i : 1 \le i < x : 2^i < N) \\
\equiv & \quad \{\; 2^0 < N \;\} \\
 & 1 \le x \;\wedge\; N \le 2^x \;\wedge\; (\forall i : 1 \le i < x : 2^i < N) \;\; .
\end{array}
$$

The formula thus derived resembles specification (0) very much: it differs from (0) only in that both occurrences of 0 have become 1's. Both formulae, therefore, are instances of the more general expression that is obtained by replacing these numbers by a fresh natural variable y:

(1) $\quad x: \;\; y \le x \;\wedge\; N \le 2^x \;\wedge\; (\forall i : y \le i < x : 2^i < N) \;\; .$

Formula (1) contains y as a free variable, so the value x specified by it depends on y: x now is the value of a *function*, applied to y. Calling this function f, we specify it as follows.

specification:

(2) Function f has type $\mathsf{Nat} \to \mathsf{Nat}$, and function value $f \cdot y$ is a solution of equation (1), for (all) natural y.

□

The type of a function is an essential ingredient of its specification. That function f has type $\mathsf{Nat} \to \mathsf{Nat}$ means, on the one hand, that $f \cdot y$ must be defined for all natural y; on the other hand, it means that $f \cdot y$ needs *not* be defined for all possible *non-natural* values of y.

For any f satisfying (2), a solution to our original problem (0) now is $f \cdot 0$. Having generalized the problem we must also generalize the previous steps towards a solution; this amounts to redoing work already done earlier, but in a more general setting, as follows. First, we observe that the range of the universal quantification in (1) is empty if $y = x$; hence, y is a solution of the equation if $N \leq 2^y$, and thus we obtain $f \cdot y = y$. Second, for the case $2^y < N$ we recalculate:

$$
\begin{array}{ll}
 & y \leq x \ \wedge \ N \leq 2^x \ \wedge \ (\forall i : y \leq i < x : 2^i < N) \\
\equiv & \quad \{ \ 2^y < N \text{, hence we have: } N \leq 2^x \Rightarrow y \neq x \ \} \\
 & y+1 \leq x \ \wedge \ N \leq 2^x \ \wedge \ (\forall i : y \leq i < x : 2^i < N) \\
\equiv & \quad \{ \text{ range split: } i = y \vee y+1 \leq i < x \text{, permitted in view of } y+1 \leq x \ \} \\
 & y+1 \leq x \ \wedge \ N \leq 2^x \ \wedge \ 2^y < N \ \wedge \ (\forall i : y+1 \leq i < x : 2^i < N) \\
\equiv & \quad \{ \ 2^y < N \ \} \\
 & y+1 \leq x \ \wedge \ N \leq 2^x \ \wedge \ (\forall i : y+1 \leq i < x : 2^i < N) \quad .
\end{array}
$$

This is an instance of formula (1) again, but with y replaced by $y+1$. If we now assume, by *induction hypothesis*, that $f \cdot (y+1)$ yields a solution for this instance, then we can use $f \cdot y = f \cdot (y+1)$ as a declaration for $f \cdot y$, for the case $2^y < N$. Combination of both cases yields a nicely recursive declaration for f:

$$
\begin{array}{lcl}
f \cdot y & = & \mathsf{if} \ \ N \leq 2^y \ \rightarrow \ y \\
 & & [] \ \ 2^y < N \ \rightarrow \ f \cdot (y+1) \\
 & & \mathsf{fi}
\end{array}
$$

The correctness of this result requires mathematical induction on the value of $N - 2^y$, which is a decreasing function of y and which has a natural and, hence, bounded value whenever $2^y < N$.

A functional program is an expression whose value has the properties required by the program specification. A functional program for our original problem, the smallest natural number x satisfying $N \leq 2^x$, is the expression $f \cdot 0$ to which the definition of f is bound by means of a where-clause, thus:

$$
\begin{array}{lllll}
f\cdot 0 & [\![& f\cdot y & = & \mathsf{if}\ \ N\leq 2^y \ \rightarrow\ \ y \\
 & & & & \ [\!]\ \ 2^y{<}N \ \rightarrow\ \ f\cdot(y{+}1) \\
 & & & & \mathsf{fi} \\
 &]\!] & & &
\end{array}
$$

This program is the functional version of the well-known *Linear Search*, here applied to the special case $N\leq 2^x$. We will discuss the general case of the Linear Search later.

* * *

That work done earlier most be redone is not as bad as it seems; the above two derivations, one for the special case and one for the general case, are very similar as the latter is a generalization of the former: the latter is obtained from the former by replacing all 0's by y and all 1's by $y{+}1$. The *only property* about 0 and 1 we have used in the special case is that 1 is the successor of 0, but this is an instance of the more general property that $y{+}1$ is the successor of y. Therefore, we need not redo the whole derivation: it suffices to generalize the result from the derivation for the special case. As long as this result only depends on general properties, that is, not on specific properties of the constants, this modus operandi is correct.

The purpose of replacing a constant by a variable is that the value specified becomes a function of the variable thus introduced. This increases our manipulative freedom, as we can now try to derive relations between the values of this function, in different points of its domain. If these relations take the shape of (so-called) *recurrence relations*, they actually define the function: that is the reason why *mathematical induction* almost always plays a role in this game.

3.2 Modularisation (i)

If we admit subexpressions like 2^y in our programs the above definition of f is fine, but if we do not (for example, because the programming language does not allow it), we have to eliminate this subexpression 2^y. This means that we have to construct an expression with value 2^y without using exponentiation.

To obtain an equivalent expression for exponentation, we need a mathematical definition of it in terms of other operations. From mathematics we know that 2^z, for any natural z, can be defined by means of a couple of recurrence relations, as follows:

$$\begin{array}{lcl} 2^0 & = & 1 \\ 2^{z+1} & = & 2 * 2^z \quad , \text{ for natural } z \quad . \end{array}$$

These are called recurrence relations because they define 2^z in a recursive way: 2^{z+1} is defined in terms of 2^z, and this is valid because z is less than $z{+}1$ (and because the naturals are *well-founded*, more about which later).

Thus, although we are only interested in 2^y, for some fixed y, we are now forced to consider 2^z for any natural z. This means that, again, we are replacing a constant, y, by a variable, z, and that we are introducing a new *function*: the value of 2^z is a function of z, which appears as a parameter here. The above recurrence relations define this function, in a recursive way.

To exploit these recurrence relations we introduce a variable g, say, for the function; we specify g by requiring it to have type $\mathsf{Nat} \to \mathsf{Nat}$ and by requiring it to satisfy:

(3) $$g{\cdot}z \;=\; 2^z \quad , \text{ for all natural } z \quad .$$

With such a g we have $2^y = g{\cdot}y$, so we may now replace the subexpressions 2^y in the definition of f by $g{\cdot}y$, and thus we obtain:

(4) $$\begin{array}{lcl} f{\cdot}y & = & \mathsf{if}\ \ N{\le}g{\cdot}y \ \to\ \ y \\ & & [\!]\ \ g{\cdot}y{<}N \ \to\ \ f{\cdot}(y{+}1) \\ & & \mathsf{fi} \end{array}$$

Finally, we must construct a declaration for g, which is easy because we only have to transscribe the above recurrence relations as a recursive declaration for our g; if we use parameter patterns this is straightforward :

(5) $$\begin{array}{lcl} g{\cdot}0 & = & 1 \\ \&\ \ g{\cdot}(z{+}1) & = & 2 * g{\cdot}z \end{array}$$

This defines g by means of multiplication, and without use of exponentiation, so we have reached our goal now.

Notice that we do not need (to know) this declaration in order to understand the above declaration (4) for f: for the latter we only need g's specification, formula (3). On the other hand, we need not know how g is used when we construct its declaration: again, its specification is all that matters. In this way, the specification of a value is the *interface* between how the value is defined and how it is used. As long as this interface is not changed, changes in its use do not affect its definition and changes in its definition do not affect its use. Thus, explicit specifications contribute to modularisation.

Formally, this role of specifications can be rendered as follows. On the one hand we have:

$$(\forall g :: (2) \Leftarrow (4) \wedge (3)) \quad ,$$

which expresses that f satisfies its specification (2), with f defined by (4) and for all g satisfying its specification (3). On the other hand we have:

$$(\forall g :: (3) \Leftarrow (5)) \quad ,$$

which expresses that every g defined by (5) satisfies specification (3). It is important to remember that this is the general shape of the proof obligations in programming: specifications must follow from declarations and specifications.

3.3 Additional Parameters (i)

The solution with the additional function g is correct, but not very *efficient*: during program execution, every replacement of $f \cdot y$ by $f \cdot (y{+}1)$ requires the evaluation of $g \cdot y$, which requires y steps. A more efficient solution is possible, though, if only we know how to exploit the following observation: evaluation of $f \cdot y$ requires the value of the expression 2^y and subsequent evaluation of $f \cdot (y{+}1)$ requires the value of 2^{y+1}, but these two values are closely related, namely by $2^{y+1} = 2 * 2^y$. Suppose we had a variable m satisfying $2^y = m$ then we would also have $2^{y+1} = 2 * m$.

Variable m can be introduced as an *additional parameter* of our function f. This is viable because the recurrence relation for 2^y *matches* the recursion pattern in the definition of f. This gives rise to a new function h of type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ and with specification:

$$(\forall m, y :: m = 2^y \Rightarrow h \cdot m \cdot y = f \cdot y) \quad .$$

So, $h \cdot m \cdot y$ has the same value as $f \cdot y$, provided $m = 2^y$. This proviso is called a *precondition* for the function application $h \cdot m \cdot y$. This specification of h gives no information about $h \cdot m \cdot y$ for m and y not satisfying the precondition: so, such cases are useless. Therefore, it is a rule of the game that the arguments of a function application must satisfy the function's preconditions. This requirement also pertains to recursive applications of the function within its own definition. The bonus, however, is that the precondition may be used to prove properties of the function.

Although parameter m is mathematically redundant, because it equals 2^y, its introduction is nevertheless meaningful because now we obtain the following functional program, in which the expression 2^y does not occur anymore:

$$
\begin{array}{lll}
h\cdot 1\cdot 0 & [\![\ h\cdot m\cdot y = & \mathsf{if}\ \ N\leq m \ \rightarrow\ \ y \\
 & & [\!]\ \ m<N \ \rightarrow\ \ h\cdot(2*m)\cdot(y+1) \\
 & & \mathsf{fi} \\
 &]\!] &
\end{array}
$$

As stated above, the obligation to show that the arguments in a function application satisfy the precondition also pertains to recursive applications of the function. So, we must show that $2*m = 2^{y+1}$, but we may use that the parameters are assumed to satisy the precondition itself, that is, we may use $m = 2^y$; moreover, we may use the preceding guard $m<N$, if necessary. Therefore, the actual proof obligation with respect to the precondition of h is:

$$2*m = 2^{y+1} \ \Leftarrow\ m = 2^y \wedge m<N \quad .$$

From a strict mathematical point of view, introducing redundant parameters is pointless, because they do not add anything new, but the effect on the efficiency of the program can be dramatic.

3.4 Exponentiation Revisited

The general of form of exponentiation is X^N, for any integer X and any natural N. For this more general case the recurrence relations are, with constant N replace by variable n:

$$
\begin{array}{lll}
X^0 & = & 1 \\
X^{n+1} & = & X * X^n \quad , \text{ for natural } n \quad .
\end{array}
$$

To implement these recurrence relations in a functional program we can now introduce a function, g, again; we require g to have type $\mathsf{Nat}\rightarrow\mathsf{Int}$, and we require it to satisfy:

(6) $\quad g\cdot n \ = \ X^n \quad , \text{ for natural } n \quad .$

With this g we have $X^N = g\cdot N$, so we have an expression for X^N if we are able to construct, in the programming language, a declaration for g. As before, this is easy: we only have to transscribe the above recurrence relations as a recursive declaration for g:

(7)
$$
\begin{array}{lll}
 & g\cdot 0 & = & 1 \\
\& & g\cdot(n+1) & = & X * g\cdot n
\end{array}
$$

Another well-known property of exponentiation is:

(8) $\quad X^{2*n} \;=\; (X^2)^n \quad .$

If we try to use this to obtain a more efficient definition for our function g we run into problems:

$$
\begin{array}{ll}
 & g\cdot(2*n) \\
= & \quad \{ \text{ specification (6) , of } g \text{ } \} \\
 & X^{2*n} \\
= & \quad \{ \text{ property (8) } \} \\
 & (X^2)^n \\
= & \quad \{ \text{ ? } \} \\
 & ? \quad .
\end{array}
$$

Because the expression $(X^2)^n$ does not match the right-hand side of g's specification (6) , we cannot complete this derivation and we cannot define $g\cdot(2*n)$ recursively in terms of g again: we are stuck. The reason is that we have obtained an expression containing a power of X^2 , whereas function g yields powers of our fixed constant X only: here we would like to substitute X^2 for X , that is, here we would like to have more manipulative freedom than is provided for in g's specification.

The way out is the observation that both X^n and $(X^2)^n$ are instances of a more general expression x^n , where x is a new variable, and that, therefore, we should replace constant X in our original formula by a variable as well. This gives rise to a new function, with one more parameter than g has; calling this new function h we obtain as its specification:

(9) $\quad h\cdot x\cdot n \;=\; x^n \quad ,$ for integer x and natural n .

Now the above derivation can be redone to completion; notice that property (8) was formulated in terms of X but, as we have used nothing particular about X , it holds for all X and, hence, also for all x :

$$
\begin{array}{ll}
 & h\cdot x\cdot(2*n) \\
= & \quad \{ \text{ specification (9) , of } h \text{ } \} \\
 & x^{2*n} \\
= & \quad \{ \text{ property (8) } \} \\
 & (x^2)^n \\
= & \quad \{ \text{ specification (9) , of } h \text{ } \}
\end{array}
$$

$$h{\cdot}x^2{\cdot}n \quad .$$

To use the equality thus obtained, $h{\cdot}x{\cdot}(2{*}n) = h{\cdot}x^2{\cdot}n$, as a proper recurrence relation, we need $n < 2{*}n$, which amounts to $1 \leq n$. So, this equality is properly recursive for positive n only; in the following declaration for h this is reflected by the use of the pattern $2{*}n{+}2$ instead of just $2{*}n$. For the odd values of the (second) parameter we simply use the recurrence relation we already had – in declaration (7) – , obtaining as recursive declaration for h :

$$\begin{array}{lll} & h{\cdot}x{\cdot}0 & = & 1 \\ \& & h{\cdot}x{\cdot}(2{*}n{+}1) & = & x * h{\cdot}x{\cdot}(2{*}n) \\ \& & h{\cdot}x{\cdot}(2{*}n{+}2) & = & h{\cdot}(x{*}x){\cdot}(n{+}1) \end{array}$$

reminder: Parameter patterns provide a shorthand notation; the above declaration is an abbreviation of this one:

$$\begin{array}{lll} h{\cdot}n & = & \mathsf{if}\ \ 0{=}n & \rightarrow & 1 \\ & & [\!]\ \ 1{\leq}n \wedge \mathit{odd}{\cdot}n & \rightarrow & x * h{\cdot}x{\cdot}(n{-}1) \\ & & [\!]\ \ 1{\leq}n \wedge \mathit{even}{\cdot}n & \rightarrow & h{\cdot}(x{*}x){\cdot}(n\,\mathsf{div}\,2) \\ & & \mathsf{fi} \end{array}$$

□

With this declaration, evaluation of $h{\cdot}x{\cdot}n$ takes only $\mathcal{O}({}^2\mathit{log}{\cdot}n)$ steps, which is a significant improvement over solutions requiring $\mathcal{O}(n)$ steps. As a result, because $X^N = h{\cdot}X{\cdot}N$, evaluation of X^N now takes $\mathcal{O}({}^2\mathit{log}{\cdot}N)$ steps.

* * *

A single expression admits of many generalizations, usually too many to make it easily predictable which generalizations will be useful. The idea of *generalization by abstraction* is that two (or more) different but yet similar expressions can often be viewed as instances – special cases – of one, more general, expression; by doing so we obtain a possibly useful generalization.

In particular, when a derivation becomes stuck because an expression is obtained that is not a (recursive or otherwise) instance of the expression under study, this may be a symptom that a generalization is needed. In principle, this implies that all derivations concerning the expression of our interest must be redone for the generalized expression. In practice, though, this is often not necessary: if the generalization is obtained by replacing a constant by a variable, and if in the derivation no properties whatsoever of that particular constant have been used, then the derivation apparently is independent of the

particular value of that constant and, hence, the derivation remains valid for the generalized case. In our previous example, for instance, the recurrence relations for X^n are valid for any constant X and, therefore, they also are valid for x^n, where x is a variable.

3.5 Recursion Revisited

Throughout the remainder of this chapter we use a somewhat more complicated example, namely the development of programs for the expression $(\Sigma i : 0 \leq i < N : X^i)$, where X and N are integer and natural constants. First, we investigate several ways to derive simple recurrence relations; next, we will study techniques to obtain relations that are less simple but give rise to more efficient programs.

* * *

Replacement of constant N by a natural variable n yields a function f, with specification:

(10) $\quad f \cdot n \;=\; (\Sigma i : 0 \leq i < n : X^i) \quad$, for natural n .

Our original expression is obtained, of course, by substitution of N back for n, so we have:

$$(\Sigma i : 0 \leq i < N : X^i) \;=\; f \cdot N \quad .$$

For this f we now derive a recursive definition, using Mathematical Induction over the domain of n, the naturals. For the base case, $n = 0$, we obtain:

$$\begin{array}{ll} & f \cdot 0 \\ = & \quad \{ \text{ specification (10), of } f \text{ } \} \\ & (\Sigma i : 0 \leq i < 0 : X^i) \\ = & \quad \{ \text{ empty range rule for } \Sigma \text{ } \} \\ & 0 \quad . \end{array}$$

Furthermore, assuming that, for some $n, 0 \leq n$, $f \cdot n$ satisfies its specification (10) – the Induction Hypothesis –, we derive for $f \cdot (n+1)$:

$$
\begin{array}{ll}
 & f\cdot(n{+}1) \\
= & \quad \{ \text{ specification (10), of } f \text{ } \} \\
 & (\Sigma i : 0{\le}i{<}n{+}1 : X^i) \\
= & \quad \{ \text{ range split } i{<}n \vee i{=}n \text{, using } 0{\le}n \text{ } \} \\
 & (\Sigma i : 0{\le}i{<}n : X^i) + X^n \\
= & \quad \{ \text{ specification (10), of } f \text{, by Induction Hypothesis } \} \\
 & f\cdot n + X^n \quad .
\end{array}
$$

Thus, we obtain as recursive declaration for f:

$$
\begin{array}{lrcl}
(11) & f\cdot 0 & = & 0 \\
\& & f\cdot(n{+}1) & = & f\cdot n + X^n
\end{array}
$$

With this declaration, evaluation of $f\cdot n$ takes $\mathcal{O}(n^2)$ steps, among which (exactly) $n*(n-1)/2$ multiplications, if we assume that every evaluation of X^n requires n multiplications.

* * *

In the last derivation we have performed a range split $i{<}n \vee i{=}n$, to decrease the size of the range. This is not the only possible way, though; it is equally well possible to use $0{=}i \vee 1{\le}i$, which gives rise to a different derivation:

$$
\begin{array}{ll}
 & f\cdot(n{+}1) \\
= & \quad \{ \text{ specification (10), of } f \text{ } \} \\
 & (\Sigma i : 0{\le}i{<}n{+}1 : X^i) \\
= & \quad \{ \text{ range split } 0{=}i \vee 1{\le}i \text{, using } 0{\le}n \text{ } \} \\
 & X^0 + (\Sigma i : 1{\le}i{<}n{+}1 : X^i) \\
= & \quad \{ \text{ dummy transformation } i{:=}i{+}1 \text{ } \} \\
 & X^0 + (\Sigma i : 0{\le}i{<}n : X^{i+1}) \\
= & \quad \{ \text{ definition of exponentiation (twice) } \} \\
 & 1 + (\Sigma i : 0{\le}i{<}n : X{*}X^i) \\
= & \quad \{ \text{ } (X*) \text{ distributes over } \Sigma \text{ } \} \\
 & 1 + X*(\Sigma i : 0{\le}i{<}n : X^i) \\
= & \quad \{ \text{ specification (10), of } f \text{, by Induction Hypothesis } \} \\
 & 1 + X*f\cdot n \quad .
\end{array}
$$

Thus, we obtain as an alternative recursive declaration for f:

$$
\begin{array}{lll}
(12) & f\cdot 0 & = & 0 \\
\& & f\cdot(n{+}1) & = & 1 + X * f\cdot n
\end{array}
$$

This recurrence relation is known as "Horner's rule". With this declaration, evaluation of $f\cdot n$ takes only $\mathcal{O}(n)$ steps, among which (exactly) n multiplications.

* * *

The above two variations show that different range splits generally yield different solutions, with different efficiencies. Hence, it is wise to try to predict – if possible – which range split will yield the best result, or to investigate several variations before making a final decision.

3.6 Modularisation (ii)

From the previous section we recall declaration (11) for our function f:

$$
\begin{array}{lll}
(11) & f\cdot 0 & = & 0 \\
\& & f\cdot(n{+}1) & = & f\cdot n + X^n
\end{array}
$$

This contains X^n as a subexpression. If X^n is not a valid expression in the programming language we have to replace it by an equivalent expression in the language, as we have studied earlier in Section 3.2. Introducing a function g for this purpose and replacing X^n by $g\cdot n$, we obtain as new declaration for f:

$$
\begin{array}{lll}
(13) & f\cdot 0 & = & 0 \\
\& & f\cdot(n{+}1) & = & f\cdot n + g\cdot n
\end{array}
$$

This is correct provided function g satisfies:

$$(14) \quad g\cdot n \;=\; X^n \;, \quad \text{for natural } n \;.$$

The simplest possible recursive declaration for g satisfying (14) is:

$$
\begin{array}{lll}
 & g\cdot 0 & = & 1 \\
\& & g\cdot(n{+}1) & = & X * g\cdot n
\end{array}
$$

This can be incorporated into the declaration for f in two ways. First, we can combine the declarations of f and g into one single declaration for both:

$$
(15) \qquad \begin{array}{lll} & f\cdot 0 & = & 0 \\ \& & f\cdot(n{+}1) & = & f\cdot n + g\cdot n \\ \& & g\cdot 0 & = & 1 \\ \& & g\cdot(n{+}1) & = & X * g\cdot n \end{array}
$$

This is useful if function g will also be applied in other expressions within the scope of f's declaration, or if we wish to treat f and g on equal footing.

Second, we can also add the declaration of g as a local declaration to the expression in which g is used, by means of a where-clause:

$$
\begin{array}{lll} & f\cdot 0 & = & 0 \\ \& & f\cdot(n{+}1) & = & f\cdot n + g\cdot n \\ & & & \mathsf{whr}\ \ g\cdot 0 = 1\ \ \&\ \ g\cdot(n{+}1) = X * g\cdot n\ \ \mathsf{end} \end{array}
$$

This variant is useful if g is not needed elsewhere and we wish to hide g within f's declaration: here g has local significance only.

Both variants of f's declaration are rather inefficient: evaluation of $f\cdot n$ requires $\mathcal{O}(n^2)$ steps and, hence, the computation of $(\Sigma i : 0{\le}i{<}N : X^i)$ by means of $f\cdot N$ will take $\mathcal{O}(N^2)$ steps. Despite their inefficiency, these declarations can nevertheless serve as a starting point for obtaining more efficient ones, as we will see later.

3.7 Additional Parameters (ii)

We recall the specification of function f from our running example:

$$
(10) \qquad f\cdot n \ = \ (\Sigma i : 0{\le}i{<}n : X^i) \ \ , \ \text{for natural } n \ \ .
$$

For the base case we still have:

$$
f\cdot 0 \ = \ 0 \ \ ,
$$

and for the other cases we can construct the following alternative derivation:

$$
\begin{array}{ll} & f\cdot(n{+}1) \\ = & \quad \{ \text{ specification (10), of } f \ \} \\ & (\Sigma i : 0{\le}i{<}n{+}1 : X^i) \\ = & \quad \{ \text{ range split } 0{=}i \vee 1{\le}i \text{, using } 0{\le}n \ \} \end{array}
$$

$$
\begin{array}{ll}
 & X^0 + (\Sigma i : 1 \le i < n+1 : X^i) \\
= & \quad \{ \text{ dummy transformation } i := i+1 \ \} \\
 & X^0 + (\Sigma i : 0 \le i < n : X^{i+1}) \quad .
\end{array}
$$

The subexpressions X^i and X^{i+1} are instances of the more general expression X^{i+m}, for any natural m. Therefore, we introduce a (more general) function g with specification:

(16) $\quad g \cdot m \cdot n \ = \ (\Sigma i : 0 \le i < n : X^{i+m}) \quad , \ \text{for natural } m, n \quad .$

Because this is a generalization, obtained by replacement of a constant 0 by a variable m, function f is easily expressed in terms of g:

(17) $\quad f \cdot n \ = \ g \cdot 0 \cdot n \quad .$

By redoing the above derivation for the general case, we obtain:

$$
\begin{array}{ll}
 & g \cdot m \cdot (n+1) \\
= & \quad \{ \text{ specification (16), of } g \ \} \\
 & (\Sigma i : 0 \le i < n+1 : X^{i+m}) \\
= & \quad \{ \text{ range split } 0 = i \vee 1 \le i \text{, using } 0 \le n \ \} \\
 & X^m + (\Sigma i : 1 \le i < n+1 : X^{i+m}) \\
= & \quad \{ \text{ dummy transformation } i := i+1 \ \} \\
 & X^m + (\Sigma i : 0 \le i < n : X^{i+1+m}) \\
= & \quad \{ \text{ specification (16), of } g \text{, by Induction Hypothesis } \} \\
 & X^m + g \cdot (1+m) \cdot n \quad .
\end{array}
$$

Thus we obtain as recursive declaration for g:

$$
\begin{array}{lll}
 & g \cdot m \cdot 0 & = & 0 \\
\& & g \cdot m \cdot (n+1) & = & X^m + g \cdot (1+m) \cdot n
\end{array}
$$

So far, so good, but if we consider the subexpression X^m in this declaration as awkward then what can we do about it? Of course, we can introduce a dedicated function for it, as we did in the previous section, but in this case the recursion pattern allows for a different approach.

First, we observe that in the recursive application of g parameter m is replaced by $1+m$; second, we already know that $X^{1+m} = X * X^m$. Suppose we had a variable y, say, satisfying $y = X^m$, then we could replace the awkward

subexpression X^m by y. We would also have $X^{1+m} = X*y$. In words: if we had a variable equal to X^m then we would have simple expressions both for X^m and for X^{1+m}.

We introduce this variable as an additional parameter y, that is assumed to be equal to X^m. This gives rise to a new function h, with one more parameter than g and with the same values as g. That the new parameter is supposed to represent X^m is formulated as a precondition. Function h can, therefore, be specified thus:

$$y = X^m \;\Rightarrow\; h{\cdot}y{\cdot}m{\cdot}n \;=\; g{\cdot}m{\cdot}n \quad , \text{ for natural } m,n \text{ and integer } y \quad .$$

Our original function f was defined, in (17), by $f{\cdot}n \;=\; g{\cdot}0{\cdot}n$, so we can now define f in terms of h by:

$$f{\cdot}n \;=\; h{\cdot}y{\cdot}0{\cdot}n \quad ,$$

provided we substitute an expression for y satisfying the precondition $y = X^0$ which, of course, amounts to $y = 1$. Hence, f can be declared in terms of h by:

$$f{\cdot}n \;=\; h{\cdot}1{\cdot}0{\cdot}n \quad .$$

When we now try to obtain a recursive declaration for h, the precondition $y = X^m$ may be exploited to obtain a simple (and efficient) expression for X^{1+m}, which is needed in the recursive application: we use $X*y$. In all other respects, function h is so similar to g that we can write down its declaration right away:

$$\begin{array}{lll} & h{\cdot}y{\cdot}m{\cdot}0 & = & 0 \\ \& & h{\cdot}y{\cdot}m{\cdot}(n{+}1) & = & y \,+\, h{\cdot}(X{*}y){\cdot}(1{+}m){\cdot}n \end{array}$$

In this declaration parameter m is not really used anymore: in the defining expression for h parameter m only occurs in the expression $1{+}m$, which is only there to provide an argument for parameter m again. So, m is superfluous now and it can be eliminated. We do so by introducing yet another new function, k, with the following specification:

$$k{\cdot}y{\cdot}n \;=\; g{\cdot}m{\cdot}n \quad , \text{ provided } y = X^m \quad , \text{ and for natural } m,n \quad .$$

Notice that m still occurs in this specification, not as a parameter but as a dummy in the universal quantification "for natural m,n".

A recursive declaration for k is easily obtained from the one for h, by omission of all occurrences of parameter m:

$$\begin{array}{lll} & k\cdot y\cdot 0 & = & 0 \\ \& & k\cdot y\cdot(n{+}1) & = & y + k\cdot(X{*}y)\cdot n \end{array}$$

Evaluation of $k\cdot y\cdot n$ takes $\mathcal{O}(n)$ steps, so this is more efficient indeed than the $\mathcal{O}(n^2)$ steps we would have obtained if we would have introduced a separate function for X^m (as we did in the previous section).

Finally, a complete declaration for function f in terms of k now is:

$$\begin{array}{lll} f\cdot n & = & k\cdot 1\cdot n \\ & & \mathsf{whr}\ \ k\cdot y\cdot 0 \quad\ = \ 0 \\ & & \quad \& \ \ k\cdot y\cdot(n{+}1) \ = \ y + k\cdot(X{*}y)\cdot n \\ & & \mathsf{end} \end{array}$$

3.8 Function Tupling

Function tupling is a program transformation technique, directed towards improvement of a program's efficiency. Mathematically speaking, the transition from a tuple of functions to a single function yielding a tuple of values is not very exciting, but the effect on the efficiency of the solution can be dramatic.

For example, in Section 3.6 we have derived the following declarations for two functions f and g:

$$\begin{array}{lll} & f\cdot 0 & = & 0 \\ \& & f\cdot(n{+}1) & = & f\cdot n + g\cdot n \\ \& & g\cdot 0 & = & 1 \\ \& & g\cdot(n{+}1) & = & X * g\cdot n \end{array} \tag{15}$$

Both f and g have the same domain – the naturals – and their declarations exhibit the same pattern of recursion: both $f\cdot 0$ and $g\cdot 0$ are defined directly and both $f\cdot(n{+}1)$ and $g\cdot(n{+}1)$ are defined recursively in terms of $f\cdot n$ and $g\cdot n$. (That $f\cdot n$ does not occur in the defining expression of $g\cdot(n{+}1)$ is harmless: we are still allowed to state that $g\cdot(n{+}1)$ is defined in terms of $f\cdot n$ and $g\cdot n$.)

We now introduce a function h, from naturals to pairs of naturals, in such a way that each value of h is a pair of values of f and g in the same point of their (common) domain. So, function h has specification:

$$h\cdot n \ = \ \langle f\cdot n, g\cdot n\rangle \ , \quad \text{for natural } n\ . \tag{18}$$

Given a function h satisfying (18), functions f and g can be easily declared in terms of h:

(19) $\quad f\cdot n \;=\; h\cdot n\cdot 0 \;\;\&\;\; g\cdot n \;=\; h\cdot n\cdot 1 \quad .$

For function h we now derive, again using Mathematical Induction over the naturals:

$$
\begin{array}{ll}
 & h\cdot 0 \\
= & \quad \{ \text{ specification (18), of } h \text{ } \} \\
 & \langle f\cdot 0, g\cdot 0\rangle \\
= & \quad \{ \text{ definition (15), of } f \text{ and } g \text{ } \} \\
 & \langle 0,1\rangle \quad ,
\end{array}
$$

and, for $n\,, 0\leq n$:

$$
\begin{array}{ll}
 & h\cdot(n{+}1) \\
= & \quad \{ \text{ specification (18), of } h \text{ } \} \\
 & \langle f\cdot(n{+}1)\,, g\cdot(n{+}1)\rangle \\
= & \quad \{ \text{ definition (15), of } f \text{ and } g \text{ } \} \\
 & \langle\, f\cdot n{+}g\cdot n\,, X{*}g\cdot n\,\rangle \\
= & \quad \{ \text{ extract } f\cdot n \text{ and } g\cdot n\text{, to prepare for the next step } \} \\
 & \langle\, x{+}y\,, X{*}y\,\rangle \;\; \mathsf{whr}\; x = f\cdot n \;\&\; y = g\cdot n \;\; \mathsf{end} \\
= & \quad \{ \text{ combine } x \text{ and } y \text{ into one single pair } \} \\
 & \langle\, x{+}y\,, X{*}y\,\rangle \;\; \mathsf{whr}\; \langle x,y\rangle = \langle f\cdot n, g\cdot n\rangle \;\; \mathsf{end} \\
= & \quad \{ \text{ specification (18), of } h\text{, by Induction Hypothesis } \} \\
 & \langle\, x{+}y\,, X{*}y\,\rangle \;\; \mathsf{whr}\; \langle x,y\rangle = h\cdot n \;\; \mathsf{end} \quad .
\end{array}
$$

Thus we obtain as recursive declaration for function h:

$$
\begin{array}{lll}
 & h\cdot 0 & = \;\; \langle 0,1\rangle \\
\& & h\cdot(n{+}1) & = \;\; \langle\, x{+}y\,, X{*}y\,\rangle \;\; \mathsf{whr}\; \langle x,y\rangle = h\cdot n \;\; \mathsf{end}
\end{array}
$$

With this declaration, the computation of $h\cdot n$, and hence of $f\cdot n$ –now defined by (19)–, takes only $\mathcal{O}(n)$ steps, among which n multiplications.

* * *

An expression like $\langle\, x{+}y\,, X{*}y\,\rangle$ is a function of the two variables x, y occurring in it, but it can be equally well viewed as the value a function applied to the single pair $\langle x,y\rangle$. That is, if we introduce a function k, declared by:

(20) $k\cdot\langle x,y\rangle \;=\; \langle\, x+y\,,\, X*y\,\rangle$,

then we can redo the above derivation for $h\cdot(n+1)$ also in this way:

$$
\begin{array}{ll}
 & h\cdot(n+1) \\
= & \quad \{ \text{ as above } \} \\
 & \langle\, f\cdot n+g\cdot n\,,\, X*g\cdot n\,\rangle \\
= & \quad \{ \text{ declaration (20) , of } k \text{ } \} \\
 & k\cdot\langle f\cdot n\,, g\cdot n\rangle \\
= & \quad \{ \text{ specification (18) , of } h \text{ , by Induction Hypothesis } \} \\
 & k\cdot(h\cdot n) \quad .
\end{array}
$$

Thus we obtain

$$h\cdot(n+1) \;=\; k\cdot(h\cdot n)$$

as a different rendering of the same recurrence relation derived above; this version lends itself for further program transformations (to be discussed later).

3.9 Divide and Conquer

More efficient recursive solutions can be obtained if it is possible to split the argument of a function into two equal parts, in such a way that on both parts the function has the same value. Efficiency is gained by having that same function value computed only once, for both parts of the argument simultaneously.

As a very simple example, here is an alternative declaration for a function g with specification $g\cdot n \;=\; X^n$; this declaration is based on yet another property of exponentiation, namely $x^{2*n} \;=\; (x^n)^2$ – compare this with property (8) used earlier – :

(21)
$$
\begin{array}{lll}
 & g\cdot 0 & = \quad 1 \\
\& & g\cdot(2*n+1) & = \quad X \,*\, g\cdot(2*n) \\
\& & g\cdot(2*n+2) & = \quad g\cdot(n+1) \,*\, g\cdot(n+1)
\end{array}
$$

This declaration still has linear time complexity: evaluation of $g\cdot n$ takes $\mathcal{O}(n)$ steps. The defining expression of $g\cdot(2*n+2)$ contains two applications of g to the same argument $n+1$: the efficiency of this declaration can be improved (significantly) if we see to it that these two applications are evaluated only once. We do so by replacing them by a fresh local variable and binding this variable to $g\cdot(n+1)$, thus:

$$
\begin{array}{lll}
 & g\cdot 0 & = & 1 \\
\& & g\cdot(2{*}n{+}1) & = & X * g\cdot(2{*}n) \\
\& & g\cdot(2{*}n{+}2) & = & y * y \ \ \mathsf{whr}\ y = g\cdot(n{+}1)\ \mathsf{end}
\end{array}
$$

The defining expression of $g\cdot(2{*}n{+}2)$ now contains only one recursive application of g, and its argument, $n{+}1$, is half as large as $2{*}n{+}2$; as a result, this declaration has logarithmic time complexity: evaluation of $g\cdot n$ now takes $\mathcal{O}(^2log\cdot n)$ steps only.

* * *

For our running example, $(\Sigma i : 0{\le}i{<}n : X^i)$, it so happens that for even n the range of the summation can be split into two equal subranges, of size $n/2$, in such a way that the whole sum can be expressed in terms of the same sums over the two subranges; as these subranges are equal, we obtain more efficient solutions. This is possible in two different ways, and we will explore both.

3.9.0 small or large

A range of the shape $i : 0{\le}i{<}2{*}n$ can be split into subranges $i : 0{\le}i{<}n$ and $i : n{\le}i{<}2{*}n$; this is the central idea in the following derivation:

$$
\begin{array}{ll}
 & f\cdot(2{*}n) \\
= & \quad \{ \text{ specification (10), of } f \ \} \\
 & (\Sigma i : 0{\le}i{<}2{*}n : X^i) \\
= & \quad \{ \text{ range split } i{<}n \vee n{\le}i \ \} \\
 & (\Sigma i : 0{\le}i{<}n : X^i) + (\Sigma i : n{\le}i{<}2{*}n : X^i) \\
= & \quad \{ \text{ dummy transformation } i := n{+}i \ \} \\
 & (\Sigma i : 0{\le}i{<}n : X^i) + (\Sigma i : 0{\le}i{<}n : X^{n+i}) \\
= & \quad \{ \ X^{n+i} = X^n{*}X^i \ ; \ (X^n{*}) \text{ distributes over } \Sigma \ \} \\
 & (\Sigma i : 0{\le}i{<}n : X^i) + X^n * (\Sigma i : 0{\le}i{<}n : X^i) \\
= & \quad \{ \text{ specification (10), of } f \text{, by Induction Hypothesis } \} \\
 & f\cdot n + X^n * f\cdot n \\
= & \quad \{ \text{ algebra } \} \\
 & (1{+}X^n) * f\cdot n \\
= & \quad \{ \text{ specification of } g \text{, to eliminate } X^n \ \} \\
 & (1{+}g\cdot n) * f\cdot n \quad .
\end{array}
$$

By combining this with the recurrence relations from declaration (15) – in Section 3.6 – and the above declaration (21) for g, we obtain:

$$
\begin{array}{lll}
 & f\cdot 0 & = & 0 \\
\& & f\cdot(2{*}n{+}1) & = & f\cdot(2{*}n) \;+\; g\cdot(2{*}n) \\
\& & f\cdot(2{*}n{+}2) & = & (1{+}g\cdot(n{+}1)) * f\cdot(n{+}1) \\
 & & & & \\
\& & g\cdot 0 & = & 1 \\
\& & g\cdot(2{*}n{+}1) & = & X \;*\; g\cdot(2{*}n) \\
\& & g\cdot(2{*}n{+}2) & = & g\cdot(n{+}1) \;*\; g\cdot(n{+}1)
\end{array}
$$

By means of tupling the two functions f and g can be combined into a single function. Formulating the specification and deriving a declaration for this function is left as one of the exercises; if carried out carefully this yields a solution in which the computation of $(\Sigma i : 0{\le}i{<}N : X^i)$ takes only $\mathcal{O}(^2 log\cdot N)$ steps.

3.9.1 even or odd

The range $i : 0{\le}i{<}2{*}n$ can also be split by distinguishing even and odd elements, as follows:

$$
\begin{array}{ll}
 & 0{\le}i{<}2{*}n \\
\equiv & \quad \{ \text{ range split } \mathit{even}\cdot i \vee \mathit{odd}\cdot i \ \} \\
 & (0{\le}i{<}2{*}n \wedge \mathit{even}\cdot i) \;\vee\; (0{\le}i{<}2{*}n \wedge \mathit{odd}\cdot i) \\
\equiv & \quad \{ \text{ dummy transformations } i := 2{*}i \text{ and } i := 2{*}i{+}1 \ \} \\
 & (0{\le}2{*}i{<}2{*}n \wedge \mathit{even}\cdot(2{*}i)) \;\vee\; (0{\le}2{*}i{+}1{<}2{*}n \wedge \mathit{odd}\cdot(2{*}i{+}1)) \\
\equiv & \quad \{ \text{ definitions of } \mathit{even} \text{ and } \mathit{odd} \ \} \\
 & 0{\le}2{*}i{<}2{*}n \;\vee\; 0{\le}2{*}i{+}1{<}2{*}n \\
\equiv & \quad \{ \text{ simplification } \} \\
 & 0{\le}i{<}n \;\vee\; 0{\le}i{<}n \quad .
\end{array}
$$

The numbers i satisfying $0{\le}i{<}2{*}n \wedge \mathit{even}\cdot i$ are the numbers of the shape $2{*}i$, for all i satisfying $0{\le}i{<}n$; therefore, we can apply a dummy transformation $i := 2{*}i$ to obtain a subrange of the shape $i : 0{\le}i{<}n$ again. Similarly, the numbers i satisfying $0{\le}i{<}2{*}n \wedge \mathit{odd}\cdot i$ are the numbers of the shape $2{*}i{+}1$, for all i satisfying $0{\le}i{<}n$; therefore, here we can apply a dummy transformation $i := 2{*}i{+}1$.

This way of splitting the range is the crux of the following derivation. If we carry out this derivation with our original expression, $(\Sigma i : 0 \le i < N : X^i)$, we will run into the need to replace X by X^2; therefore, we generalize the problem by replacing constant X by a variable x, that is, we study a function g with specification:

(22) $\quad g \cdot x \cdot n \;=\; (\Sigma i : 0 \le i < n : x^i) \;$, for natural n .

As before, we still have $g \cdot x \cdot 0 = 0$; for the even values of n we derive:

$$
\begin{array}{ll}
 & g \cdot x \cdot (2*n) \\
= & \quad \{ \text{ specification (22), of } g \text{ } \} \\
 & (\Sigma i : 0 \le i < 2*n : x^i) \\
= & \quad \{ \text{ range split } even \cdot i \vee odd \cdot i \text{ , and dummy transformations } \} \\
 & (\Sigma i : 0 \le i < n : x^{2*i}) + (\Sigma i : 0 \le i < n : x^{2*i+1}) \\
= & \quad \{ \text{ } (x*) \text{ over } \Sigma \text{ } \} \\
 & (\Sigma i : 0 \le i < n : x^{2*i}) + x * (\Sigma i : 0 \le i < n : x^{2*i}) \\
= & \quad \{ \text{ property of exponentiation } \} \\
 & (\Sigma i : 0 \le i < n : (x^2)^i) + x * (\Sigma i : 0 \le i < n : (x^2)^i) \\
= & \quad \{ \text{ specification (22), of } g \text{, by Induction Hypothesis } \} \\
 & g \cdot x^2 \cdot n + x * g \cdot x^2 \cdot n \\
= & \quad \{ \text{ algebra } \} \\
 & (1+x) * g \cdot x^2 \cdot n \quad .
\end{array}
$$

For the odd values of n we can use the same derivation that gave rise to declaration (12) in Section 3.5; thus, we obtain the following efficient declaration for g:

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

3.10 Exercises

Met Fibonacci, bijvoorbeeld, in allerlei varianten...

en:

$$\begin{array}{lll} & f\cdot 0 & = & 1 \\ \& & f\cdot(n{+}1) & = & X * f\cdot n \end{array}$$

$$\begin{array}{lll} & g\cdot 0 & = & 1 \\ \& & g\cdot(n{+}1) & = & g\cdot n * X \end{array}$$

Bewijs nu dat $f\cdot n = g\cdot n$, voor alle n; welke algebraïsche eigenschappen van $*$ zijn hierbij nodig?

De functie *fusc*.

Fibolucci.

Chapter 4

On Specifications and Proofs

Examine all things; retain what is good.
Paulus

4.0 Specifications and Programs

A predicate on the value domain Ω is a boolean function on Ω: for every element in Ω the predicate has a value that is either true or false. Elements in Ω for which the predicate has value true also are said to *satisfy* the predicate.

Every predicate partitions Ω into two disjoint subsets, namely: those values that satisfy the predicate and those values that do not satisfy the predicate. If satisfaction of the predicate is all we are interested in, then the predicate also induces an equivalence relation on Ω: all elements satisfying the predicates are equivalent – they are *equally good* –, and all elements not satisfying the predicate are equivalent too – they are *equally bad* –.

* * *

Different expressions may require different amounts of time to compute their values, even if these values are equal. So, efficiency is property of expressions, not of their values.

4.1 Recursion and Induction

Over hoe je $R{\cdot}X \Leftarrow Q{\cdot}X$ bewijst als $Q{\cdot}X$ van de vorm $X = F{\cdot}X$ is en $R{\cdot}X$ van de vorm $(\forall c :: R{\cdot}c{\cdot}X)$, waarbij het domein van c well-founded is. Productiviteit; meer hierover bij de behandeling van oneindige lijsten.

4.2 Preconditions

Over specificaties van de vorm $P \Rightarrow R \cdot X$. Bewijsregel voor guarded selections; bewijsregel voor recursieve functiedefinities.

4.3 The shape of derivations

Over hoe $R \cdot X \Leftarrow Q \cdot X$ in de praktijk wordt gerealiseerd. Wens/feit-stappen.

4.4 Syntactic versus semantic typing

Chapter 5

Efficiency Considerations

5.0 Introduction

The execution of a computer program requires computation time and storage space. Different programs (for the same problem) may require different amounts of time and space for their execution: some programs are more *efficient* than others. The uses of time and space are not necessarily related: one program may need less time but more space than another program.

To be able to judge the efficiency of a functional program we must understand how it is executed. So, we will discuss this and we will develop rules to determine the time complexity of functional programs. In addition we will shortly discuss usage of storage space. To start with we only consider the applicative algebra, as defined in Chapter 1. The extensions to the formalism, as introduced in Chapters 2 and 6, are not true extensions but can be considered syntactic sugar and, hence, can be largely ignored.

A functional program is an expression, and executing it means calculating its value, more precisely, calculating a suitable *representation* of its value. The adjective "suitable" is crucial here: every expression is a representation of its own value, but this can hardly be considered "suitable". For instance, as value of the expression $2+3$ we prefer 5 over the expression itself.

A particular requirement is that all expressions having the *same* value give rise to the *same* representation of their common value; that is, if (syntactically) different expressions have the same value then their evaluations should yield the same answer. As a simple example, if evaluation of $2+3$ yields "5" then evaluation of $4+1$ and $3*4-7$ must yield "5" too. In short, for every class of equivalent expressions the representation of their common value must be *unique*. Such a unique common representation of equivalent expressions is

called the *normal form* of these expressions.

Another requirement is the requirement of computability: to evaluate an an expression means to calculate its value in a *finite* number of steps. Unfortunately but unavoidably, however, our functional programming language[0] contains expressions that cannot be evaluated in a finite number of steps. As a consequence, such expressions do not (and cannot) have normal forms: by definition, expression evaluation is only possible for those expressions that have normal forms.

In what follows we confine our attention to the set of expressions that have normal forms. This does not mean that expressions without normal forms are useless. On the contrary: they often occur as subexpressions in larger expressions, and these larger expressions may have normal forms. Integer functions on the naturals and infinite lists with integer elements, for example, generally have no normal forms – they represent infinitely many values – but each individual function value or list element often can be calculated and, hence, has a normal form.

5.1 Reduction and Normal Forms

The normal forms may be expressions themselves: in every class of equivalent expressions we can choose one particular expression and define this expression as the normal form for that class. We will do so in a particular way, as explained below, based on the notion of *reduction*.

Evaluation of an expression can take place according to the following general (and abstract) scheme: as long as it is not a normal form the expression is repeatedly *transformed*, by means of some rule, into another but equivalent expression. This process may or may not terminate, but if it terminates the resulting expression is the normal form of the original expression.

Whether or not this general evaluation scheme terminates depends both on the properties of the formalism and on the transformation rule used. One particularly interesting kind of transformation is known as reduction, which is common to all implementations of functional languages.

The main calculation rule in the applicative algebra is the Rule of Instantiation, which we recall from Chapter 1.

Rule of Instantiation: If x is declared by:

$$x \cdot y \cdot z \;=\; E \quad ,$$

[0] like any other general-purpose programming language

then for all expressions A, B, not containing y, z as free variables, x satisfies:

$$x \cdot A \cdot B \;=\; E(y := A)(z := B) \quad .$$

□

This rule expresses equality of the expressions $x \cdot A \cdot B$ and $E(y := A)(z := B)$ and this equality is a true mathematical equality; in particular, it is symmetric and in calculational derivations we use this equality in either direction: we may equally well replace $x \cdot A \cdot B$ by $E(y := A)(z := B)$ as the other way round.

The Rule of Instantiation may be applied to any *subexpression* of an expression at hand: both $x \cdot A \cdot B$ and $E(y := A)(z := B)$ may occur as subexpressions of a larger expression and may, within this larger expression, be replaced by each other. This is a direct consequence of the fact that the rule expresses true equality: it is just Leibniz's principle of substitution of equals for equals.

If, however, the Rule of Instantiation is applied exclusively in one direction, namely from left to right, we speak of *reduction*: a single replacement of a subexpression $x \cdot A \cdot B$ by $E(y := A)(z := B)$, on account of the rule, is called a (single) *reduction step*, and the process of performing zero or more such reduction steps is simply called *reduction*.

If x is a declared variable with 2 parameters then we call any expression of the shape $x \cdot A \cdot B$ a *redex*[1], and we call any expression *reducible* that contains at least one redex as a subexpression. So, a redex is a subexpression matching the left-hand side of an (applicable) declaration. An expression containing no redexes at all is called *irreducible*.

Generally, an expression may contain more than one redexes and, hence, generally, an expression admits several different reduction steps. In that case a *reduction strategy* is needed to prescribe which of the possible reduction steps is actually performed. Notice, however, that a reduction step is a special application of the Rule of Instantiation and that, hence, reduction does not affect the value of the expression: independent of the reduction strategy, reduction transforms expressions into equivalent expressions.

An expression without redexes is irreducible. We now define these irreducible expressions to be the normal forms. Regardless of the reduction strategy used, the reduction process terminates if an expression is obtained without redexes and, by definition, this expression now is the normal form of the expression with which the reduction process started.

[1] just short for *red*ucible sub*ex*pression

It is (a corollary of) a theorem in the λ-calculus that every class of equivalent expressions contains *at most one* irreducible expression. Hence, if it exists, the normal form of a class of expressions is unique.

5.2 Lazy Evaluation

What remains is the problem of termination, that is, the choice of a reduction strategy. One particular (and deterministic) reduction strategy is called *normal-order reduction*. Another theorem in the λ-calculus states that, for any expression that has a normal form, normal-order reduction yields that normal form in finitely many steps. So, normal-order reduction can be used to evaluate every expression that can be evaluated at all.

Normal-order reduction is based on the following observations. As in the previous subsection, we assume that x is a declared variable with 2 parameters, defined thus:

$$x \cdot y \cdot z \;=\; E \quad .$$

As before, the following observations are valid for declared variables with any number – zero or more – of parameters. We distinguish the following cases:

0. An expression of the shape $x \cdot A \cdot B$ is a redex, so it is not a normal form. In addition, the argument expressions A and B may be reducible themselves. Whatever reduction steps are applied to either A or B, however, such steps will not change the general shape $x \cdot A \cdot B$; hence, no reduction step applied to either A or B only will transform the expression into normal form. Normal order reduction, therefore, prescribes that $x \cdot A \cdot B$ is reduced first (by replacing it by $E(y{:=}A)(z{:=}B)$): sooner or later this step has to be taken anyhow, whereas it remains to be seen whether reduction of A or B are really necessary.

1. An expression of the shape $F \cdot A \cdot B$, where F is not a declared variable with 2 (or fewer) parameters, is not itself a redex but both F and A and B may be reducible. In the same vein as in the previous case, reduction of F may (or may not) turn the expression into a redex, whereas reduction of A or B certainly will not turn the whole expression into a redex. Therefore, normal reduction prescribes that F is reduced first, as long as this is possible, and as long as the previous case does not arise.

2. An expression of the shape $F \cdot A \cdot B$, where F is not a declared variable with (at most) 2 parameters and where F is irreducible, is not itself a redex and it will never become one: the only redexes, if any, now occur in A or B, and reduction steps to A or B will not change this situation. Only in this case reduction of A and B is meaningful (and necessary). Normal order reduction prescribes that A is evaluated first, but actually this is irrelevant: in this case, both A and B must be fully evaluated and these two processes are completely independent. As a matter of fact, A and B may even be evaluated *concurrently*.

3. An expression of the shape $x \cdot A$ is a function application with fewer argument expressions than the number of x's parameters: generally, this is considered an error and the reduction process is terminated. Similarly, evaluation of a function without argument expressions at all is generally considered erroneous.

4. The above rules also apply to expressions of the shape $x \cdot A \cdot B \cdot C \cdots$, that is, with more argument expressions than x has parameters. Because such an expression is parsed as $(((x \cdot A) \cdot B) \cdot C) \cdots$ it contains $(x \cdot A) \cdot B$ as a redex, to be treated according to the above rules.

remark: Substitution is a laborious operation. The replacement of $x \cdot A \cdot B$ by $E(y := A)(z := B)$ is, therefore, usually implemented as the replacement of $x \cdot A \cdot B$ by E whr $y = A$ & $z = B$ end, which is not so laborious. This way, the argument expressions are bound to the parameters by means of where-clauses. The net effect is that the actual substitutions $y := A$ and $z := B$ are *postponed* until they are really needed.
□

Normal-order reduction is demand driven; it is also known as *lazy evaluation*, because a subexpression is only evaluated to the extent its value is really needed to calculate the value of the main expression in which that subexpression occurs. Some subexpressions will not be evaluated at all, namely if the value of the main expression does not depend on their values. Other subexpressions will only be evaluated *partially*, namely if the value of the main expression only depends on parts of their values. Such subexpressions need not even have normal forms: it suffices that the relevant parts of their values can be computed.

simple examples: A recursively defined function of type Nat → Nat has

no normal form but any application of such function to a natural number does have a normal form. Thus, the value of the whole function can not be computed in finitely many steps, but the value of the function in any point of its domain can be computed.

A function f defined by:

$$f \cdot x \;=\; 5 \quad ,$$

has a constant value that is independent of its parameter. Hence, normal order reduction of $f \cdot E$ will yield 5 without performing any reduction step to expression E; consequently, E may be any expression, with or without normal form.

For more elaborate examples we refer to Section 5.7.

□

5.3 Subexpression Sharing

5.3.0 duplicated subexpressions

We do not expect any "intelligence" from the implementation. In particular, we do not expect the implementation to detect that the same subexpression appears in different places, which would be an opportunity to save reduction steps. The simplest example of this is an expression like $E+E$: as soon as one of the E's has been evaluated its value might also be substituted for the other E, but we do not expect the implementation to do so.

A programmer can, however, make such equality of subexpressions explicit by replacing them by a (fresh) variable and binding the common subexpression to that variable in a where-clause, thus:

$$E+E \;=\; x+x \;\; \mathsf{whr}\; x=E \;\mathsf{end} \quad .$$

With standard normal order reduction this does not help, however, because every occurrence of x will be simply replaced by a copy of E again. Therefore, in actual implementations of functional languages, a modified version of normal order reduction is used. Instead of substituting (a copy of) E for x, followed by (partially or completely) reducing this copy of E, the implementation reduces the single occurrence of E *inside* the where-clause. This continues until an expression is obtained that, when substituted for x, creates a redex in the surrounding expression. In the case of integer addition, as in the example $E+E$, this means that E will be fully evaluated first before it is substituted for x. As a result, any subsequent occurrences of the variable will not give rise to new evaluations of the subexpression anymore.

For the example $x+x$ whr $x=E$ end the difference between standard and modified normal order reduction can be illustrated by the following calculations. Standard normal order reduction proceeds as follows – where we assume E's value to be 5 –:

$$
\begin{array}{ll}
 & x+x \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{substitution}\ \} \\
 & E+x \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{evaluation of } E \text{ (possibly taking many steps)}\ \} \\
 & 5+x \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{substitution}\ \} \\
 & 5+E \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{evaluation of } E \text{ (possibly taking many steps)}\ \} \\
 & 5+5 \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{addition}\ \} \\
 & 10 \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{where-clause elimination}\ \} \\
 & 10 \ \ ,
\end{array}
$$

whereas modified normal order reduction proceeds in this way:

$$
\begin{array}{ll}
 & x+x \ \ \mathsf{whr}\ x=E\ \mathsf{end} \\
= & \quad \{\ \text{evaluation of } E \text{ (possibly taking many steps)}\ \} \\
 & x+x \ \ \mathsf{whr}\ x=5\ \mathsf{end} \\
= & \quad \{\ \text{substitution}\ \} \\
 & 5+x \ \ \mathsf{whr}\ x=5\ \mathsf{end} \\
= & \quad \{\ \text{substitution}\ \} \\
 & 5+5 \ \ \mathsf{whr}\ x=5\ \mathsf{end} \\
= & \quad \{\ \text{addition}\ \} \\
 & 10 \ \ \mathsf{whr}\ x=5\ \mathsf{end} \\
= & \quad \{\ \text{where-clause elimination}\ \} \\
 & 10 \ \ .
\end{array}
$$

Thus, repeated evaluation of expression E is avoided; this is particularly attractive if evaluation of E requires very many steps indeed.

5.3.1 the advantage of postponed substitutions

The use of where-clauses to bind argument expressions to parameters, so as to streamline substitutions, also enables expression sharing for parameters. That is, by implementing a reduction like:

$$x \cdot A \cdot B \;:=\; E(y := A)(z := B) \quad ,$$

as:

$$x \cdot A \cdot B \;:=\; E \;\mathsf{whr}\; y = A \;\&\; z = B \;\mathsf{end} \quad ,$$

we obtain sharing of the subexpressions A and B for free; after all, now these subexpressions are not substituted in multitude anymore, for all, possibly many, occurrences (in E) of the parameters y and z, respectively.

As we will see in the examples, the positive effect of this on the time complexity of a functional program can be dramatic.

5.3.2 a few special cases

An expression may contain a function application $f \cdot E$, say. As before, multiple occurrences of this function application can be reduced to a single occurrence, by introducing a variable y, say, and binding this variable to $f \cdot E$ in a where-clause. Of course, this is only possible for multiple applications of function f to one and the same argument expression E, like this:

$$\begin{array}{ll} & \cdots\; f \cdot E \;\cdots\; f \cdot E \;\cdots \\ = & \quad \{ \text{ abstraction } \} \\ & \cdots\; y \;\cdots\; y \;\cdots \;\; \mathsf{whr}\; y \;=\; f \cdot E \;\mathsf{end} \quad . \end{array}$$

An even more special case arises if we apply this technique to the definition of function f itself, in the case $f \cdot x$ is defined recursively in terms of $f \cdot x$:

$$\begin{array}{ll} & f \cdot x \;=\; \cdots\; f \cdot x \;\cdots\; f \cdot x \;\cdots \\ = & \quad \{ \text{ abstraction } \} \\ & f \cdot x \;=\; y \;\; \mathsf{whr}\; y \;=\; \cdots\; y \;\cdots\; y \;\cdots \;\mathsf{end} \quad . \end{array}$$

Again, this seemingly innocent transformation may have a dramatic effect on the time complexity of the program. In Chapter 11 we will encounter applications of this, where, for a function from infinite lists to infinite lists, this transformation improves the time complexity of the function from quadratic to linear.

5.4 Which Types Have Normal Forms?

It is highly relevant to know which expressions can evaluated in a finite amount of time. For any given expression this depends on the expression's *type*. Generally, an expression has a normal form if (and only if) its value represents a finite amount of information; this means that its value is one out of at most countably many possible values. Thus, the type of the expression may be at most countably infinite.

Using this as the general guiding principle we obtain that (expressions of) the following types have normal forms. We call such types the *normal types*:

0. the basic types: boolean, natural, and integer;
1. the composite types: tuples and finite lists, provided their elements have normal types as well.

Because the elements of tuples and finite lists are required to have normal types, instead of just basic types, the class of the normal types is actually quite large: it comprises recursive datatypes that can be defined by means of tuples and finite lists, such as *finite trees*.

Generally, functions and infinite lists have no normal forms. As stated earlier, functions and infinite lists are nevertheless useful, because each of their values (or elements) still may be computable.

5.5 The Time Complexity of Function Applications

We now consider function applications, that is, expressions of the shape $f \cdot A \cdot B$, where f is a function with 2 parameters and where A and B are expressions. As before, this discussion is applicable to any number, zero or more, of parameters and, as before, we use 2 parameters as prototype.

The time needed to compute $f \cdot A \cdot B$ now is: the time needed to evaluate A and B, as far as needed, plus the time needed to compute $f \cdot A \cdot B$ itself, under the assumption that A and B have been sufficiently evaluated.

This distinction is meaningful because the time needed to compute $f \cdot A \cdot B$ itself generally depends on the values of the expressions A and B but not on their shapes: their evaluation times can be factored out and what remains we call the evaluation time of the function application proper, independently of the shape of the arguments.

Hence, the evaluation time of a function application proper generally is a function of the values of the application's arguments. If the function is

defined directly – that is, non-recursively –, this evaluation time simply is the time needed to evaluate the function's defining expression, as a function of the values of the function's parameters. If the function's definition is recursive, however, the analysis becomes more complicated.

Let function f be defined by:

$$f \cdot x \cdot y \;=\; E \quad ,$$

where E, f's defining expression, may contain occurrences of then parameters x and y and E may also contain recursive applications of f.

We now introduce a function φ, with this intended interpretation:

$$\varphi \cdot x \cdot y \;=\; \text{“the evaluation time of } f \cdot x \cdot y \text{”} \quad .$$

Because of the shape of f's definition, φ must satisfy:

$$\varphi \cdot x \cdot y \;=\; \text{“the evaluation time of } E \text{”} \quad ,$$

and the evaluation time of E can be determined

5.6 Space Utilization

Voor zover daar iets zinnigs over valt te zeggen. Het wordt in ieder geval summier. Voorbeelden waaruit blijkt dat het niet vanzelf gaat, zoals met een “accumulating parameter”, zoals faculteit: de naieve oplossing gebruikt nog steeds O(n) geheugen.

5.7 Examples

5.7.0 reduction

With H declared by:

$$H \;=\; H \quad ,$$

we have that H can be reduced to H, and this is the only possible reduction step. Hence, (repeated) reduction of H does not terminate and, as a consequence, H has no normal form.

As a slightly less trivial example, with W declared by:

$$W \cdot x \;=\; x \cdot x \quad ,$$

we have that $W \cdot W$ can be reduced to $W \cdot W$, and this is the only possible reduction step. Hence, reduction of $W \cdot W$ does not terminate and, as a consequence, $W \cdot W$ has no normal form.

These simple examples show that not every class of expressions in the applicative algebra contains normal forms. Notice that, in these simple examples, this conclusion is independent of the reduction strategy: the expressions H and $W \cdot W$ contain a single redex only and, hence, the reduction strategy is irrelevant.

Notice that W itself is irreducible and, hence, W is its own normal form. Apparently, expressions can be composed from normal forms and yet have no normal forms themselves; for example: $W \cdot W$.

With H as above and with f declared by:

$$f \cdot x \;=\; 5 \quad ,$$

we have that $f \cdot H = 5$ and because normal-order reduction will apply the declaration for f first, evaluation of $f \cdot H$ will indeed produce 5. A reduction strategy that would give preference to reduction of subexpression H, on the other hand, would give rise to a computation that does not terminate: reduction of H transforms $f \cdot H$ into itself, without progress.

This last example illustrates lazy evaluation: because the value of $f \cdot H$ does not depend on subexpression H, there is no need to evaluate H; because of the danger of non-termination it is even better not to evaluate a subexpression as long as its value is not needed.

5.7.1 the effect of expression sharing

5.8 human versus mechanical computation

Here is a nice example illustrating that mechanical evaluation by means of reduction is quite a different game than how we, human beings, calculate. We consider functions tw ("twice") and sc ("successor") defined by:

$$\begin{array}{lcll} tw \cdot f \cdot x & = & f \cdot (f \cdot x) & \text{, for all } f, x \;; \\ sc \cdot x & = & 1 + x & \text{, for natural } x \;. \end{array}$$

What, then, is the value of the expression:

$$tw \cdot tw \cdot tw \cdot sc \cdot 0 \quad ?$$

5.8.0 a calculation by hand

The problem is not so much how to calculate the value of the given expression, but more how to do so without too much effort. Efficiency can be gained if we succeed in enlarging our manipulative possibilities, for example, by first formulating some (hopefully) useful properties of tw and sc . First, we observe that the definition of tw can be rephrased as:

(0) $\quad tw \cdot f \;=\; f \circ f \quad ,$

and, second, we obtain, by the instantiation $f := tw$, as a special case:

(1) $\quad tw \cdot tw \;=\; tw \circ tw \quad .$

Now we calculate:

$$
\begin{array}{ll}
 & tw \cdot tw \cdot tw \cdot sc \cdot 0 \\
= & \quad \{ \ (1) \ \} \\
 & (tw \circ tw) \cdot tw \cdot sc \cdot 0 \\
= & \quad \{ \text{ function composition } \} \\
 & tw \cdot (tw \cdot tw) \cdot sc \cdot 0 \\
= & \quad \{ \ (1) \ \} \\
 & tw \cdot (tw \circ tw) \cdot sc \cdot 0 \\
= & \quad \{ \ (0) \text{ , with } f := tw \circ tw \ \} \\
 & ((tw \circ tw) \circ (tw \circ tw)) \cdot sc \cdot 0 \\
= & \quad \{ \text{ continued function composition } \} \\
 & tw^4 \cdot sc \cdot 0 \\
= & \quad \{ \text{ continued function composition } \} \\
 & tw^3 \cdot (tw \cdot sc) \cdot 0 \\
= & \quad \{ \ (0) \text{ , with } f := sc \ \} \\
 & tw^3 \cdot (sc^2) \cdot 0 \\
= & \quad \{ \text{ idem, with } f := sc^2 \ \} \\
 & tw^2 \cdot (sc^4) \cdot 0 \\
= & \quad \{ \text{ idem, with } f := sc^4 \ \} \\
 & tw \cdot (sc^8) \cdot 0 \\
= & \quad \{ \text{ idem, with } f := sc^8 \ \}
\end{array}
$$

$$sc^{16} \cdot 0$$

At this stage in the calculation the properties of sc become relevant. For all natural n and x we have:

$$sc^{n} \cdot x \;=\; n + x \quad .$$

Hence, we obtain $sc^{16} \cdot 0 = 16$, which solves the problem:

$$tw \cdot tw \cdot tw \cdot sc \cdot 0 \;=\; 16 \quad .$$

The brevity of the above calculation is mainly due to the fact that we can exploit the associativity of function composition in whatever way we see fit. In addition, we are neither restricted to the use of reduction nor to any particular order: we can replace any subexpression by any equivalent expression we like. Of course, we do have to keep in mind that our target is to be reached in finitely many steps. In the above example, we needed only 4 steps to obtain an expression, namely $((tw \circ tw) \circ (tw \circ tw)) \cdot sc \cdot 0$, from which it is clear that the calculation will not derail: once this expression is obtained the problem is essentially solved.

5.8.1 a calculation by machine

Here we show a calculation as it is performed by a machine using the normal-order reduction strategy: in each step the left-most subexpression that matches the left-hand side of one of the definitions is replaced by the corresponding right-hand side. This calculation takes 54 steps; each step is an application of the definition of either tw or sc or $+$. Subexpression sharing is not used here; in this example sharing would save only very few steps.

Observe the difference with the manual calculation, but do not verify this calculation in its minute details: the whole point of using a machine is that we should not need to verify its output, once we have verified its program. Actually, the following calculation has indeed been produced by means of a computer:

$$\begin{aligned}
& (((tw \cdot tw) \cdot tw) \cdot sc) \cdot 0 \\
= \{tw\} \;\; & ((tw \cdot (tw \cdot tw)) \cdot sc) \cdot 0 \\
= \{tw\} \;\; & ((tw \cdot tw) \cdot ((tw \cdot tw) \cdot sc)) \cdot 0 \\
= \{tw\} \;\; & (tw \cdot (tw \cdot ((tw \cdot tw) \cdot sc))) \cdot 0 \\
= \{tw\} \;\; & (tw \cdot ((tw \cdot tw) \cdot sc)) \cdot ((tw \cdot ((tw \cdot tw) \cdot sc)) \cdot 0) \\
= \{tw\} \;\; & ((tw \cdot tw) \cdot sc) \cdot (((tw \cdot tw) \cdot sc) \cdot ((tw \cdot ((tw \cdot tw) \cdot sc)) \cdot 0))
\end{aligned}$$

$$
\begin{array}{ll}
= \{tw\} & (tw\cdot(tw\cdot sc))\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)) \\
= \{tw\} & (tw\cdot sc)\cdot((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))) \\
= \{tw\} & sc\cdot(sc\cdot((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))) \\
= \{sc\} & 1+(sc\cdot((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))) \\
= \{sc\} & 1+(1+((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))) \\
= \{tw\} & 1+(1+(sc\cdot(sc\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))))) \\
= \{sc\} & 1+(1+(1+(sc\cdot(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))))) \\
= \{sc\} & 1+(1+(1+(1+(((tw\cdot tw)\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))))) \\
= \{tw\} & 1+(1+(1+(1+((tw\cdot(tw\cdot sc))\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))))) \\
= \{tw\} & 1+(1+(1+(1+((tw\cdot sc)\cdot((tw\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))) \\
= \{tw\} & 1+(1+(1+(1+(sc\cdot(sc\cdot((tw\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(sc\cdot((tw\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+((tw\cdot sc)\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(sc\cdot(sc\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot((tw\cdot tw)\cdot sc))\cdot 0)))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(((tw\cdot tw)\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot 0))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot(tw\cdot sc))\cdot(((tw\cdot tw)\cdot sc)\cdot 0))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot sc)\cdot((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot 0))))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot(sc\cdot(((tw\cdot tw)\cdot sc)\cdot 0)))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot 0))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot sc)\cdot(((tw\cdot tw)\cdot sc)\cdot 0))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(((tw\cdot tw)\cdot sc)\cdot 0)))))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot(tw\cdot sc))\cdot 0)))))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot sc)\cdot((tw\cdot sc)\cdot 0)))))))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot(sc\cdot((tw\cdot sc)\cdot 0)))))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot((tw\cdot sc)\cdot 0)))))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+((tw\cdot sc)\cdot 0)))))))))))))) \\
= \{tw\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot(sc\cdot 0)))))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(sc\cdot 0)))))))))))))) \\
= \{sc\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+0)))))))))))))))
\end{array}
$$

$$
\begin{array}{ll}
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+1)))))))))))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+2)))))))))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+3)))))))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+4)))))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+5))))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+(1+6)))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+7))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+(1+(1+8))))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+(1+9)))))) \\
=\{+\} & 1+(1+(1+(1+(1+(1+10))))) \\
=\{+\} & 1+(1+(1+(1+(1+11)))) \\
=\{+\} & 1+(1+(1+(1+12))) \\
=\{+\} & 1+(1+(1+13)) \\
=\{+\} & 1+(1+14) \\
=\{+\} & 1+15 \\
=\{+\} & 16 \quad .
\end{array}
$$

Evidently, this calculation takes much more steps than our manual solution and, because of the size of the intermediate expressions, this calculation also requires more storage space. Fortunately, computers are much faster than we are, and they have huge memories.

5.9 Exercises

1. Determine the time complexity, as a function of natural n, of the function application $fib \cdot n$, where function fib is defined recursively by:

$$
\begin{array}{lcl}
fib \cdot 0 & = & 0 \\
fib \cdot 1 & = & 1 \\
fib \cdot (n+2) & = & fib \cdot n + fib \cdot (n+1)
\end{array}
$$

2. Idem, where fib is defined by:

$$
\begin{array}{lcl}
fib \cdot 0 & = & 0 \\
g \cdot 0 & = & 1 \\
fib \cdot (n+1) & = & g \cdot n \\
g \cdot (n+1) & = & fib \cdot n + g \cdot n
\end{array}
$$

3. Determine the time complexity, as a function of natural n, of the function application $h \cdot n$, where function h is defined recursively by:

$$\begin{array}{lcl} h \cdot 0 & = & \langle\, 0\,,1\,\rangle \\ h \cdot (n{+}1) & = & \langle\, y\,, x{+}y\,\rangle \ \mathsf{whr}\ \langle\, x\,, y\,\rangle \ = \ h \cdot n \ \mathsf{end} \end{array}$$

4. Prove $tw^n \cdot f \ = \ f^{2^n}$, for $0 \leq n$, with tw as defined in Section 5.8.

5. What types does function tw have? What is the type of the expression $tw \cdot tw$? Does this follow from tw's types (and the general typing rules for functions)?

6. Meer oefeningen ...

2000.12.8 2013.4.15 (correction of a few simple errors)

Chapter 6

Finite Lists

6.0 What are Lists?

A list is a datastructure whose elements are linearly ordered; also, the elements of a list usually (but not necessarily) have the same type. A list can be *finite* or *infinite*. A separate chapter will be devoted to the theory needed to reason about infinite lists. To illustrate both the differences and the correspondences we include, however, the definition of infinite lists and a few simple examples in this chapter.

6.0.0 the list primitives

Both the datatypes of finite lists and the datatypes of infinite lists can be defined in terms of a small set of primitive operators which we call the *list primitives*. The properties of these list primitives are the only ones needed to derive all other list properties.

The list primitives – with their suggested pronunciation – are:

a constant $[\,]$ –"empty"–;

a binary operator $\triangleright$ –"cons"–;

a function *ise* –"is empty"–;

a function *tl* –"tail"–.

We adopt the convention that $\triangleright$ is *right-binding*, that is, expressions like $x \triangleright y \triangleright z$ must be read thus:

$$x \triangleright y \triangleright z \;=\; x \triangleright (y \triangleright z) \quad .$$

The list primitives are postulated to have the following properties, for all x, y (in Ω), and for natural n:

$$\begin{array}{lcl} ise\cdot[\,] & = & \mathsf{true} \\ ise\cdot(x \triangleright y) & = & \mathsf{false} \\ \\ (x \triangleright y)\cdot 0 & = & x \\ (x \triangleright y)\cdot(n{+}1) & = & y\cdot n \\ \\ tl\cdot(x \triangleright y) & = & y \end{array}$$

These properties are sufficient to prove other simple properties, like, for example, for all x, y, u, v:

$$x \triangleright y \;\neq\; [\,] \;\;,$$

and:

$$x \triangleright y \;=\; u \triangleright v \quad\equiv\quad x = u \wedge y = v \;\;.$$

The latter property states that $\triangleright$ is *injective* and, hence, it has an *inverse*: from a composite value $x \triangleright y$ the constituents x and y can be retrieved, namely by means of $(\cdot 0)$ and tl:

$$(x \triangleright y)\cdot 0 \;=\; x \quad\wedge\quad tl\cdot(x \triangleright y) \;=\; y \;\;.$$

In most other functional programming languages two dedicated functions, hd and tl, are used for this purpose, with:

$$hd\cdot(x \triangleright y) \;=\; x \quad\wedge\quad tl\cdot(x \triangleright y) \;=\; y \;\;,$$

but instead of hd we use $(\cdot 0)$ because it allows a simple generalization to $(\cdot n)$, which we will use for general *element selection*. Later, when we discuss the (so-called) *take-and-drop calculus*, we will generalize tl in a similar way.

6.0.1 finite lists

For any type B and for natural n, the datatype $\mathcal{L}_n(B)$ is the type of all *lists of length* n and with elements of type B. Also, the datatype $\mathcal{L}_*(B)$ is the type of all *finite lists* with elements of type B; this just is the *union* of $\mathcal{L}_n(B)$, over all n. Formally, these types are defined as follows, for all t, $t \in \Omega$, and for natural n:

$$
\begin{array}{lcl}
t \in \mathcal{L}_0(B) & \equiv & t = [\,] \\
t \in \mathcal{L}_{n+1}(B) & \equiv & (\exists b, s : b \in B \wedge s \in \mathcal{L}_n(B) : t = b \triangleright s) \\
t \in \mathcal{L}_*(B) & \equiv & (\exists n :: t \in \mathcal{L}_n(B))
\end{array}
$$

In words, the one-and-only list of length 0 is $[\,]$, and every list of length $n{+}1$ is of the shape $b \triangleright s$, where b is an element and s is a list of length n. We call $[\,]$ the *empty list*, whereas we call lists of the shape $b \triangleright s$ *composite lists*. A finite list is a list of length n, for *some* natural n.

6.0.2 infinite lists

For any type B and for natural n, the datatype $\mathcal{P}_n(B)$ is the type of all *listoids of order* n and with elements of type B. Also, the datatype $\mathcal{L}_\infty(B)$ is the type of all *infinite lists* with elements of type B; this just is the *intersection* of $\mathcal{P}_n(B)$, over all n. Formally, these types are defined as follows, for all t, $t \in \Omega$, and for natural n:

$$
\begin{array}{lcl}
t \in \mathcal{P}_0(B) & \equiv & \mathsf{true} \\
t \in \mathcal{P}_{n+1}(B) & \equiv & (\exists b, s : b \in B \wedge s \in \mathcal{P}_n(B) : t = b \triangleright s) \\
t \in \mathcal{L}_\infty(B) & \equiv & (\forall n :: t \in \mathcal{P}_n(B))
\end{array}
$$

In words, every value (in Ω) is a listoid of order 0, and every listoid of order $n{+}1$ is of the shape $b \triangleright s$, where b is an element and s is a listoid of order n. An infinite list is a listoid of order n, for *all* natural n.

6.0.3 all lists are listoids

A finite list of length n has *exactly* n elements. A listoid of order n has *at least* n elements. Hence, every list of length n is a listoid of order n as well. This is reflected in the above definitions of $\mathcal{L}$ and $\mathcal{P}$. From these definitions follows:

$$(\forall n :: \mathcal{L}_n(B) \subseteq \mathcal{P}_n(B)) \quad .$$

As a consequence, everything we say or prove about listoids can also be said or proved about finite lists.

6.0.4 additonial notation

Whenever no confusion is possible or when we are only interested in the structure and not in the type of the elements, we may wish to leave the element type implicit: in such a case we omit the element type and abbreviate $\mathcal{L}_n(B)$ to $\mathcal{L}_n$ and $\mathcal{L}_*(B)$ to $\mathcal{L}_*$. Similarly, we may abbreviate $\mathcal{P}_n(B)$ and $\mathcal{L}_\infty(B)$ to $\mathcal{P}_n$ and $\mathcal{L}_\infty$.

By definition, $\mathcal{L}_0$ has only one element, namely $[\,]$. The elements of $\mathcal{L}_1(B)$ are the values of the shape $b \triangleright [\,]$, for b, $b \in B$. The elements of $\mathcal{L}_2(B)$ are the values of the shape $b \triangleright c \triangleright [\,]$, for b, c, $b, c \in B$, and so on: the finite lists are the values *constructed from* $[\,]$ and $\triangleright$.

When writing out all elements of a list explicitly, we sometimes use the following abbreviations:

$$\begin{array}{rcl} [b] & = & b \triangleright [\,] \\ [b,c] & = & b \triangleright c \triangleright [\,] \\ [b,c,d] & = & b \triangleright c \triangleright d \triangleright [\,] \\ & \vdots & \end{array}$$

In particular, $[b]$ is the *singleton list* $b \triangleright [\,]$ whose one-and-only element is b. Also, we have properties like $[b,c,d] = b \triangleright [c,d]$.

Notice that a finite list always "ends with $[\,]$"; for example, $b \triangleright c \triangleright d \triangleright x$ is a list if and only if x is a list, in particular if $x = [\,]$, and an expression like:

$$b \triangleright c \triangleright d$$

is *not* a list (unless d itself is a list), whereas $[b,c,d]$ is a list because this expressions is an abbreviation of $b \triangleright c \triangleright d \triangleright [\,]$.

6.0.5 element selection

We recall that $\triangleright$ has the following properties, for all x, y in Ω and for all natural n:

$$\begin{array}{lcl} (x \triangleright y) \cdot 0 & = & x \\ (x \triangleright y) \cdot (n+1) & = & y \cdot n \end{array}$$

As a result, we can use function $(\cdot n)$ to select the element at position n from any listoid that has enough elements. The element at position n is the element with n predecessors, so, for example, the element at position 0 is the first element. More precisely, for natural n and listoid s with $s \in \mathcal{P}_{n+1}$, we have that $s \cdot n$ is s's element at position n.

example:

$$\begin{array}{lcl} [b,c,d]\cdot 0 & = & b \;\; , \\ [b,c,d]\cdot 1 & = & c \;\; , \text{ and:} \\ [b,c,d]\cdot 2 & = & d \;\; . \end{array}$$

Also, we have $(b \triangleright c \triangleright d \triangleright x)\cdot 3 \;=\; x\cdot 0$. Whether or not $(b \triangleright c \triangleright d \triangleright x)\cdot 3$ is meaningful depends on x: it is meaningful if $x \in \mathcal{P}_1$, and it is certainly not meaningful if $x = [\,]$.

□

6.0.6 some properties

The following properties are useful for reasoning about lists in a calculational way, for all b, s, t, n:

$$b \triangleright s \in \mathcal{L}_{n+1}(B) \;\equiv\; b \in B \wedge s \in \mathcal{L}_n(B)$$

$$b \triangleright s \in \mathcal{P}_{n+1}(B) \;\equiv\; b \in B \wedge s \in \mathcal{P}_n(B)$$

$$b \triangleright s \in \mathcal{L}_*(B) \;\equiv\; b \in B \wedge s \in \mathcal{L}_*(B)$$

$$b \triangleright s \in \mathcal{L}_\infty(B) \;\equiv\; b \in B \wedge s \in \mathcal{L}_\infty(B)$$

$$t \in \mathcal{L}_*(B) \;\equiv\; t = [\,] \;\vee\; (\exists b, s : b \in B \wedge s \in \mathcal{L}_*(B) : t = b \triangleright s)$$

$$t \in \mathcal{L}_\infty(B) \;\equiv\; (\exists b, s : b \in B \wedge s \in \mathcal{L}_\infty(B) : t = b \triangleright s)$$

$$t \in \mathcal{L}_n(B) \;\equiv\; t \in \mathcal{L}_n(\Omega) \;\wedge\; (\forall i : i < n : t\cdot i \in B)$$

$$t \in \mathcal{L}_\infty(B) \;\equiv\; t \in \mathcal{L}_\infty(\Omega) \;\wedge\; (\forall i :: t\cdot i \in B) \;\; .$$

The latter two properties express that the requirements that a value is a list and that all its elements have type B can be proved separately.

6.1 Functions and Lists

6.1.0 list parameters

Functions can have parameters that are (assumed to be) lists. As an example, we consider a function f , of type $\mathcal{L}_*(\mathsf{Int}) \to \mathsf{Int}$, and specified by:

$$f\cdot t \;=\; (\Sigma i : i < n : t\cdot i) \;\; , \text{ for natural } n \text{ and } t \in \mathcal{L}_n(\mathsf{Int}) \;\; .$$

Because every finite list either is $[\,]$ or is of the shape $b \triangleright s$, for some integer b and finite list s, we confine ourselves to these two cases. Because the length of s is smaller than the length of $b \triangleright s$, we may use Mathematical Induction. We derive:

$$
\begin{array}{ll}
 & f\cdot[\,] \\
= & \quad \{ \text{ specification of } f\text{, using } [\,]\in\mathcal{L}_0 \ \} \\
 & (\Sigma i : i<0 : t\cdot i) \\
= & \quad \{ \text{ empty range } \} \\
 & 0 \quad ,
\end{array}
$$

from which we conclude that $f\cdot[\,] = 0$ satisfies f's specification. Also, for integer b and $s\in\mathcal{L}_n$ –and, hence, $b \triangleright s \in \mathcal{L}_{n+1}$– we derive:

$$
\begin{array}{ll}
 & f\cdot(b \triangleright s) \\
= & \quad \{ \text{ specification of } f\text{, using } b \triangleright s \in \mathcal{L}_{n+1} \ \} \\
 & (\Sigma i : i<n+1 : (b \triangleright s)\cdot i) \\
= & \quad \{ \text{ range split } 0=i \vee 1\leq i\text{, to prepare for the next step } \} \\
 & (b \triangleright s)\cdot 0 + (\Sigma i : i<n : (b \triangleright s)\cdot(i+1)) \\
= & \quad \{ \text{ property of } \triangleright \text{ (twice) } \} \\
 & b + (\Sigma i : i<n : s\cdot i) \\
= & \quad \{ \text{ specification of } f\text{, by Induction Hypothesis, using } s\in\mathcal{L}_n \ \} \\
 & b + f\cdot s \quad ,
\end{array}
$$

from which we conclude that $f\cdot(b \triangleright s) = b + f\cdot s$ satisfies f's specification too.

By collecting these results into a single declaration we obtain the following recursive declaration for f:

$$
\begin{array}{lll}
 & f\cdot[\,] & = & 0 \\
\& & f\cdot(b \triangleright s) & = & b + f\cdot s
\end{array}
$$

6.1.1 list values

Functions can have values that are (required to be) lists. As a simple example, we consider function f, of type $\mathsf{Nat} \to \mathcal{L}_*(\mathsf{Int})$, and specified by:

$$
f\cdot n \in \mathcal{L}_n(\mathsf{Int}) \ \wedge \ (\forall i : i<n : f\cdot n\cdot i = 7) \quad , \text{ for natural } n \quad .
$$

In words this specification states that $f \cdot n$ is a list of length n and that all its elements are 7.

If it were not for the requirement that $f \cdot n$ is a list, as formalised by the conjunct $f \cdot n \in \mathcal{L}_n(\mathsf{Int})$, a perfectly acceptable declaration for f would be:

$$f \cdot n \cdot m \;=\; 7 \quad ,$$

because this certainly satisfies the requirement $f \cdot n \cdot i = 7$ for all natural n, i with $i < n$. Now, however, we must take into account that $f \cdot n$ must be a list, and, therefore, we proceed as follows.

Because of the case distinction in the definition of $\mathcal{L}_n$, we distinguish 0 from the positive naturals and, first, we derive:

$$\begin{array}{ll} & f \cdot 0 \in \mathcal{L}_0 \\ \equiv & \quad \{ \text{ definition of } \mathcal{L}_0 \ \} \\ & f \cdot 0 = [\,] \quad , \end{array}$$

which leaves no further room for choices: apparently, $f \cdot 0$ must be $[\,]$ and, fortunately, the second conjunct of f's specification is met as well, because the range of its universal quantification is empty in this case.

Furthermore, for any natural n we derive:

$$\begin{array}{ll} & f \cdot (n{+}1) \in \mathcal{L}_{n+1}(\mathsf{Int}) \\ \equiv & \quad \{ \text{ definition of } \mathcal{L}_{n+1} \ \} \\ & (\exists b, s : b \in \mathsf{Int} \wedge s \in \mathcal{L}_n(\mathsf{Int}) : \ f \cdot (n{+}1) = b \triangleright s \,) \end{array}$$

So, we *must* define $f \cdot (n{+}1)$ as $b \triangleright s$, for some appropriately chosen integer b and list s of length n. To investigate what further properties b and s must have we try to prove the second conjunct from f's specification:

$$\begin{array}{ll} & (\forall i : i < n{+}1 : \ f \cdot (n{+}1) \cdot i = 7 \,) \\ \equiv & \quad \{ \text{ proposal for } f \cdot (n{+}1) \ \} \\ & (\forall i : i < n{+}1 : \ (b \triangleright s) \cdot i = 7 \,) \\ \equiv & \quad \{ \text{ range split, to prepare for the next step } \} \\ & (b \triangleright s) \cdot 0 \;=\; 7 \;\wedge\; (\forall i : i < n : \ (b \triangleright s) \cdot (i{+}1) = 7 \,) \\ \equiv & \quad \{ \text{ properties of } \triangleright \ \} \end{array}$$

$$
\begin{array}{ll}
 & b = 7 \ \wedge \ (\forall i : i < n : s \cdot i = 7) \\
\equiv & \quad \{ \text{ specification of } f \text{, by Induction Hypothesis } \} \\
 & b = 7 \ \wedge \ (\forall i : i < n : s \cdot i = f \cdot n \cdot i) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & b = 7 \ \wedge \ s = f \cdot n \quad .
\end{array}
$$

By Induction Hypothesis we also have $f \cdot n \in \mathcal{L}_n(\mathsf{Int})$; hence, $s = f \cdot n$ is an acceptable choice. Thus, by combining the two cases, we obtain the following recursive declaration for f :

$$
\begin{array}{lll}
 & f \cdot 0 & = \ [\,] \\
\& & f \cdot (n{+}1) & = \ 7 \triangleright f \cdot n
\end{array}
$$

6.1.2 a more concise presentation: the "$\triangleright$-trick"

In the last example, the only way to define $f \cdot (n{+}1)$ as a list of length $n{+}1$ is by means of an expression of the shape $b \triangleright s$; knowing this, we can distinguish the cases 0 and $i{+}1$ in the specification of the list's elements right from the start. Thus, we obtain a more concise rendering of essentially the same derivation:

$$
\begin{array}{ll}
 & f \cdot (n{+}1) \cdot 0 \\
= & \quad \{ \text{ specification of } f \ \} \\
 & 7 \\
= & \quad \{ \text{ property of } \triangleright \text{ , explanation follows } \} \\
 & (7 \triangleright ?) \cdot 0 \quad .
\end{array}
$$

The question mark in the formula $(7 \triangleright ?) \cdot 0$ denotes a so-called "don't care" : a subexpression whose value is irrelevant because the equality only depends on the left argument, 7 , of $\triangleright$ and not on that subexpression. We use such a question mark whenever we do not wish to commit ourselves to a premature choice for the subexpression it represents: in a later stage of the derivation we may badly need the freedom we have here.

We now continue for natural i , $i < n$:

$$
\begin{array}{ll}
 & f \cdot (n{+}1) \cdot (i{+}1) \\
= & \quad \{ \text{ specification of } f \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & 7 \\
= & \quad \{ \text{ specification of } f \text{, by Induction Hypothesis } \} \\
 & f \cdot n \cdot i \\
= & \quad \{ \text{ property of } \triangleright \ \} \\
 & (? \triangleright f \cdot n) \cdot (i{+}1) \quad .
\end{array}
$$

In this case, only the *second* argument of $\triangleright$ is relevant and we don not care about the first argument.

The two (incomplete) expressions $7 \triangleright ?$ and $? \triangleright f \cdot n$ can now be unified to $7 \triangleright f \cdot n$ and the results of the last two derivations can be combined into:

$$(\forall i : i < n{+}1 : \ f \cdot (n{+}1) \cdot i = (7 \triangleright f \cdot n) \cdot i \ ) \quad .$$

This can be strengthened – this is just Leibniz – to:

$$f \cdot (n{+}1) \ = \ 7 \triangleright f \cdot n \quad .$$

Thus, we obtain the same recursive declaration for f as in the previous subsection:

$$
\begin{array}{lll}
 & f \cdot 0 & = \ [\,] \\
\& & f \cdot (n{+}1) & = \ 7 \triangleright f \cdot n
\end{array}
$$

6.1.3 a taste of infinity

Infinite lists have no lengths and, therefore, we cannot use Mathematical Induction on their lengths. We can, however, use Mathematical Induction on the *domain* of infinite lists, which is the naturals. As a simple example, we consider the problem how to define an infinite list all whose elements have the value 7. If we call this list x its specification is:

$$x \in \mathcal{L}_\infty(\mathsf{Int}) \quad \wedge \quad (\forall i :: x \cdot i = 7 \ ) \quad .$$

As is the case with the composite finite lists, infinite lists can only be constructed by means of the operator $\triangleright$, so we have to introduce $\triangleright$ into the game one way or another. We do so by means of a derivation that resembles the derivation in the previous subsection very much, by distinguishing the cases 0 and $i{+}1$ in x's specification:

$$
\begin{array}{ll}
 & x\cdot 0 \\
= & \quad \{ \text{ specification of } x \text{ } \} \\
 & 7 \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (7 \triangleright ?)\cdot 0 \quad ,
\end{array}
$$

and for natural i:

$$
\begin{array}{ll}
 & x\cdot(i{+}1) \\
= & \quad \{ \text{ specification of } x \text{ } \} \\
 & 7 \\
= & \quad \{ \text{ specification of } x\text{, by Induction Hypothesis: } i < i{+}1 \text{ } \} \\
 & x\cdot i \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (? \triangleright x)\cdot(i{+}1) \quad .
\end{array}
$$

The two (incomplete) expressions $7 \triangleright ?$ and $? \triangleright x$ can now be unified to $7 \triangleright x$ and the results of the last two derivations can be combined into:

$$(\forall i :: x\cdot i = (7 \triangleright x)\cdot i \,) \quad ,$$

which can be strengthened to the following valid declaration:

$$x \;=\; 7 \triangleright x \quad .$$

As a result of the above calculations this declaration implies $(\forall i :: x\cdot i = 7\,)$. That it also implies $x \in \mathcal{L}_\infty(\mathsf{Int})$ remains as an additional proof obligation. Because, by definition, $x \in \mathcal{L}_\infty(\mathsf{Int})$ is equivalent to $(\forall n :: x \in \mathcal{P}_n(\mathsf{Int})\,)$, it suffices to prove the latter, which we do by Mathematical Induction:

$$
\begin{array}{ll}
 & x \in \mathcal{P}_0(\mathsf{Int}) \\
\equiv & \quad \{ \text{ definition of } \mathcal{P}_0 \text{ } \} \\
 & \mathsf{true} \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & x \in \mathcal{P}_{n+1}(\mathsf{Int}) \\
\equiv & \quad \{ \text{ declaration of } x \text{ } \}
\end{array}
$$

$$
\begin{array}{ll}
 & 7 \triangleright x \in \mathcal{P}_{n+1}(\mathsf{Int}) \\
\equiv & \quad \{ \text{ property of } \mathcal{P}_{n+1} \ \} \\
 & 7 \in \mathsf{Int} \ \wedge \ x \in \mathcal{P}_n(\mathsf{Int}) \\
\equiv & \quad \{ \ 7 \in \mathsf{Int} \ \} \\
 & x \in \mathcal{P}_n(\mathsf{Int}) \quad .
\end{array}
$$

6.2 More Operators on Finite Lists

6.2.0 length

The prefix operator $\#$ – "length" or "size" – has type $\mathcal{L}_n \to \mathsf{Nat}$, for all natural n. For all s we have:

$$s \in \mathcal{L}_n \ \Rightarrow \ \# s = n \quad .$$

A recursive definition of $\#$ is easily derivable from this specification (and the definition of $\mathcal{L}_n$):

$$
\begin{array}{lll}
 & \#[\,] & = \ 0 \\
\& & \#(b \triangleright s) & = \ \# s + 1
\end{array}
$$

6.2.1 cat

By means of operator $\triangleright$ a list of length $n{+}1$ is constructed from an element and a list of length n. From two lists s and t of lengths m and n, a list of length $m{+}n$ can be constructed by means of *concatenation*. For this purpose we use a binary operator $+\!\!+$: the result of concatenating s and t is $s +\!\!+ t$.

Thus, the specification of $+\!\!+$ is, for all s, t and natural m, n:

$$
\begin{array}{ll}
 & s \in \mathcal{L}_m \ \wedge \ t \in \mathcal{L}_n \\
\Rightarrow & \\
 & s +\!\!+ t \in \mathcal{L}_{m+n} \ \wedge \ (\forall i : i < m : (s +\!\!+ t) \cdot i = s \cdot i \,) \ \wedge \\
 & (\forall i : i < n : (s +\!\!+ t) \cdot (m{+}i) = t \cdot i \,) \quad .
\end{array}
$$

A recursive declaration for $+\!\!+$ is:

$$
\begin{array}{lll}
 & [\,] +\!\!+ t & = \ t \\
\& & (b \triangleright s) +\!\!+ t & = \ b \triangleright (s +\!\!+ t)
\end{array}
$$

Concatenation has other nice algebraic properties: $+\!\!+$ has $[\,]$ as its identity element and $+\!\!+$ is associative. We refer to Section 6.3 for an overview.

6.2.2 reverse

Function rev maps a finite list to its *reverse*, that is, function rev has type $\mathcal{L}_n \to \mathcal{L}_n$, for all natural n, and:

$$(\forall i : i \leq n : rev \cdot s \cdot i = s \cdot (n-i)) \quad , \text{ for } s \in \mathcal{L}_{n+1} \quad .$$

Notice that this formula does not specify $rev \cdot [\,]$, but also notice that this is unnecessary because the empty list has no elements: from the type requirement $\mathcal{L}_0 \to \mathcal{L}_0$ alone, because $\mathcal{L}_0$ has only one element, it follows – we have no choice here – that:

$$rev \cdot [\,] \;=\; [\,] \quad .$$

Other properties of rev that follow from its specification are:

$$\begin{array}{lcl} rev \cdot (b \triangleright t) & = & rev \cdot t \mathbin{+\!\!+} [b] \\ rev \cdot [b] & = & [b] \\ rev \cdot (s \mathbin{+\!\!+} t) & = & rev \cdot t \mathbin{+\!\!+} rev \cdot s \end{array}$$

6.2.3 snoc

Occasionally, we have need of an operator complementary to $\triangleright$, which we call $\triangleleft$ – "snoc" –. Whereas, for example:

$$[b, c, d] \;=\; b \triangleright [c, d] \quad ,$$

we now also have:

$$[b, c, d] \;=\; [b, c] \triangleleft d \quad ,$$

and so on.

Whereas $b \triangleright s \;=\; [b] \mathbin{+\!\!+} s$ we now also have $s \triangleleft b \;=\; s \mathbin{+\!\!+} [b]$.

6.2.4 map

An important operation is to construct, from a function and a list, the list of function values obtained by application of the function to all elements of the given list. For this purpose we introduce an operator $\bullet$ – "map" –. For function f of type $B \to V$ and list s of type $\mathcal{L}_n(B)$, the type of $f \bullet s$ is $\mathcal{L}_n(V)$, and we have, for all natural n:

$$(\forall i : i < n : (f \bullet s) \cdot i = f \cdot (s \cdot i)) \quad , \text{ for } s \in \mathcal{L}_n \quad .$$

From this specification the following properties can be derived:

$$
\begin{array}{lcl}
f\bullet[\,] & = & [\,] \\
f\bullet(b\triangleright t) & = & f\cdot b \triangleright f\bullet t \\
f\bullet[b] & = & [\,f\cdot b\,] \\
f\bullet(s+\!\!+t) & = & f\bullet s +\!\!+ f\bullet t
\end{array}
$$

6.3 Summary of Finite List Properties

In the following summary r, s, and t denote finite lists, whereas b and c denote elements. All properties in this summary are universally quantified over their free variables.

6.3.0 cons and snoc

$$
\begin{array}{lcll}
b\triangleright t & \neq & [\,] & \\
s\triangleleft c & \neq & [\,] & \\
(b\triangleright t)\cdot 0 & = & b & \\
(s\triangleleft c)\cdot n & = & c & ,\ n=\#s \\
(b\triangleright t)\cdot(i{+}1) & = & t\cdot i & ,\ i<\#t \\
(s\triangleleft c)\cdot i & = & s\cdot i & ,\ i<\#s \\
b\triangleright[\,] & = & [b] & \\
[\,]\triangleleft c & = & [c] & \\
b\triangleright s\ =\ c\triangleright t & \equiv & b=c\ \wedge\ s=t & \\
s\triangleleft b\ =\ t\triangleleft c & \equiv & s=t\ \wedge\ b=c &
\end{array}
$$

6.3.1 cat

$$
\begin{array}{lcll}
(s+\!\!+t)\cdot i & = & s\cdot i & ,\ i<\#s \\
(s+\!\!+t)\cdot(n{+}i) & = & t\cdot i & ,\ i<\#t\ \wedge\ n=\#s \\
[\,]+\!\!+t & = & t & \\
s+\!\!+[\,] & = & s & \\
(b\triangleright s)+\!\!+t & = & b\triangleright(s+\!\!+t) & \\
s+\!\!+(t\triangleleft c) & = & (s+\!\!+t)\triangleleft c & \\
[b]+\!\!+t & = & b\triangleright t & \\
s+\!\!+[c] & = & s\triangleleft c & \\
r+\!\!+(s+\!\!+t) & = & (r+\!\!+s)+\!\!+t & \\
s+\!\!+t\ =\ [\,] & \equiv & s=[\,]\ \wedge\ t=[\,] &
\end{array}
$$

6.3.2 rev

$$
\begin{array}{lcll}
rev\cdot s \in \mathcal{L}_n & , & \text{for } s \in \mathcal{L}_n & \\
rev\cdot s\cdot i & = & s\cdot(n-i) & , \quad i \le n \wedge n+1 = \#s \\
rev\cdot[\,] & = & [\,] & \\
rev\cdot[b] & = & [b] & \\
rev\cdot(b \triangleright s) & = & rev\cdot s \triangleleft b & \\
rev\cdot(t \triangleleft c) & = & c \triangleright rev\cdot t & \\
rev\cdot(s +\!\!+ t) & = & rev\cdot t +\!\!+ rev\cdot s & \\
rev\cdot(rev\cdot s) & = & s &
\end{array}
$$

6.3.3 length

$$
\begin{array}{lcl}
\#[\,] & = & 0 \\
\#[b] & = & 1 \\
\#(b \triangleright t) & = & \#t + 1 \\
\#(s \triangleleft c) & = & \#s + 1 \\
\#(s +\!\!+ t) & = & \#s + \#t
\end{array}
$$

6.3.4 map

$$
\begin{array}{lcll}
(f\bullet s)\cdot i & = & f\cdot(s\cdot i) & , \quad i < \#s \\
f\bullet[\,] & = & [\,] & \\
f\bullet[b] & = & [f\cdot b] & \\
f\bullet(b \triangleright t) & = & f\cdot b \triangleright f\bullet t & \\
f\bullet(s \triangleleft c) & = & f\bullet s \triangleleft f\cdot c & \\
f\bullet(s +\!\!+ t) & = & f\bullet s +\!\!+ f\bullet t & \\
f\bullet(rev\cdot s) & = & rev\cdot(f\bullet s) &
\end{array}
$$

Chapter 7

Representations

7.0 Introduction

Very often one data type is used to *represent* – or *implement* – another data type. For the sake of efficiency, the representing data type may contain more details than the represented data type: these details are irrelevant for its intended use but may be necessary – or just: convenient – for the efficient implementation of the operations on that data type. Because of this, a represented data type is also called an *abstract data type*, whereas a representing data type is also called a *concrete data type*.

The relation between a concrete data type and its represented abstract data type is fixed by a function mapping the elements of the concrete data type to the elements of the abstract data type. This function is called an *abstraction function*; the use of a function is appropriate here because a function is the proper device to achieve the desired abstraction from whatever detail is considered irrelevant.

simple examples: Finite sets can be represented by finite lists, simply by enumeration of the set's elements in a list; the detail irrelevant for its use, in this case, is the order in which the set's element appear in the list; hence, the abstraction function will map all permutations of the same list to the same set, thus effectively abstracting from the differences between these permutations.

Natural numbers can be represented by lists of digits, according to the rules of the positional number system. In base 10, for example, the sequence of digits "137" represents the number "one-hundred-and-thirty-seven", and so does the sequence of digits "000137". The ab-

straction function maps sequences of digits to natural numbers, in a way that ignores the irrelevant leading zeroes and that does away with the actual base of the number system used.

□

Because every element of the abstract datatype must be representable by at least one element of the concrete datatype, the abstraction function must be chosen in such a way that every abstract element is the value of the function for at least one concrete element: the abstraction function must be *surjective*.

Conversely, the abstraction function needs not be injective, because the representation needs not be *unique*. It is perfectly acceptable that the same abstract value is represented by more than one element of the concrete data type; sometimes, this reduncany even is an advantage, because it may give rise to more efficient programs.

In addition, if operations on the abstract data type have been defined then "corresponding" operations must be defined on the concrete data type, so as to *implement* these abstract operations in terms of the concrete data type. Hence, representing an abstract data type by a concrete one involves not only the choice of a suitable abstraction function but also the implementation of the abstract type's operations by means of concrete operations.

Data type representations may be nested: an abstract type may itself serve as the (concrete) representation type for an even more abstract data type. Thus, the adjectives "abstract" and "concrete" are always relative ones: a data type is concrete if it represents another, more abstract, data type, but it may itself be viewed as an abstract data type when its own representation is considered.

7.1 Abstraction Functions

By definition, a set X represents a set V if and only if a surjective function φ exists, of type $X \to V$. Via φ the elements of V are represented by elements of X: element x in X represents $\varphi \cdot x$ in V. Because φ is surjective, every element of V is thus represented by at least one element of X.

Function φ is called the abstraction function, because it maps the concrete data type to the abstract data type. In particular, it abstracts from different representations of the same element: x and y represent the same value, if $\varphi \cdot x = \varphi \cdot y$.

Because of φ's surjectivity, quantified formulae over V may be rephrased as quantified formulae over X, by means of dummy transformation and pro-

vided the quantor involved is based on an idempotent operator. For example, for predicate P on V, universal quantifications may be rewritten thus:

$$
\begin{array}{ll}
 & (\forall v : v{\in}V : P{\cdot}v) \\
\equiv & \quad \{ \ \varphi \text{ is surjective: dummy transformation } \} \\
 & (\forall x : x{\in}X : P{\cdot}(\varphi{\cdot}x)) \quad .
\end{array}
$$

If φ is injective (and, hence, bijective) equalities like this also hold for quantors based on non-idempotent operators, like Σ.

7.2 Operators

With a data type generally several different operators are defined, of several different types. Operators are constants and functions involving the data type.

Here we only investigate the simpler ones, which provides enough insight so as to enable the implementation of operators of arbitrary signatures.

remark: To really capture the general underlying patterns would require an excursion into Category Theory, but this falls ouside the scope of this text.

□

In the following subsections we assume that set V is represented by set X, via abstraction function φ. The requirements formulated in these subsections are in fact *specifications* for the implementation of the abstract operators: these requirements can be used for the development of definitions for the corresponding concrete operators.

7.2.0 constants

The implementation of a constant D of type V as a constant of type X should satisfy this requirement, which follows directly from the definition of the abstraction function:

$$D = \varphi{\cdot}C$$

Viewed as a specification this specifies C in terms of D and φ; because C only occurs as argument in a function application, C needs not be unique.

7.2.1 functions to a data type

For some type U, a function g, of type $U \to V$, is implemented as a function f, of type $U \to X$, if and only if:

(0) $\quad (\forall u : u \in U : g \cdot u = \varphi \cdot (f \cdot u)) \quad .$

By abstraction from dummy u this can also be formulated more succintly as a function composition:

(1) $\quad g = \varphi \circ f \quad .$

Both (0) and (1) specify f in terms of g and φ; again, this will usually not specify f uniquely.

7.2.2 functions on a data type

For some type W, a function g, of type $V \to W$, is implemented as a function f, of type $X \to W$, if and only if:

(2) $\quad (\forall x : x \in X : g \cdot (\varphi \cdot x) = f \cdot x) \quad ,$

which can be rewritten as a function composition too:

(3) $\quad g \circ \varphi = f \quad .$

7.2.3 functions on and to a data type

By combining the results from the previous two subsections we obtain that function g, of type $V \to V$, is implemented as a function f, of type $X \to X$, if and only if:

(4) $\quad (\forall x : x \in X : g \cdot (\varphi \cdot x) = \varphi \cdot (f \cdot x)) \quad ,$

which, when rewritten as a function composition, takes this shape:

$$g \circ \varphi = \varphi \circ f \quad .$$

7.2.4 congruences

Every function induces an equivalence relation on its domain. In particular, abstraction function φ induces an equivalence relation $\simeq$ on its domain X. This is the "represents the same value"-relation, defined thus:

(5) $\quad x \simeq y \;\equiv\; \varphi\cdot x = \varphi\cdot y$, for all x, y in X .

Specifications of implementations, like (2) and (4), restrict the space of solutions: generally, not every function in the concrete world represents a function in the abstract world.

For example, not every function of type $X \to W$ implements a function of type $V \to W$. This is demonstrated as follows, where we assume that f implements g according to (2), and where x and y in X are assumed to satisfy $\varphi\cdot x = \varphi\cdot y$:

$$
\begin{array}{ll}
 & f\cdot x \\
= & \quad \{ \text{ specification (2) } \} \\
 & g\cdot(\varphi\cdot x) \\
= & \quad \{ \varphi\cdot x = \varphi\cdot y \} \\
 & g\cdot(\varphi\cdot y) \\
= & \quad \{ \text{ specification (2) } \} \\
 & f\cdot y \;\; ,
\end{array}
$$

from which we conclude that function f in $X \to W$ implements a function in $V \to W$ (if and) only if f satisfies this necessary condition:

$$(\forall x, y : x, y \in X : \varphi\cdot x = \varphi\cdot y \Rightarrow f\cdot x = f\cdot y) \quad .$$

Reformulated in terms of equivalence relation $\simeq$ this becomes:

$$(\forall x, y : x, y \in X : x \simeq y \Rightarrow f\cdot x = f\cdot y) \quad .$$

This is just a condition of *congruence*: apparently, implementations of functions must be compatible with $\simeq$.

Similarly, a function f in $X \to X$ implements a function in $V \to V$ if and only if it satisfies this congruence requirement:

$$(\forall x, y : x, y \in X : x \simeq y \Rightarrow f\cdot x \simeq f\cdot y) \quad .$$

7.3 Number Representations

In every day life, natural numbers are represented in the decimal number system, in which strings of decimal digits are used. This system is *positional*: the value of a digit depends on its position in the string.

The abstraction function maps strings of digits to natural numbers. A recursive definition for the abstraction function can be obtained, by means of the following observation. The natural number 4137 equals $10 * 413 + 7$; therefore, the number represented by the string "4137" equals ten times the number represented by its prefix "413" plus its last digit "7". This way, the number represented by a string can be defined in terms of the string's prefix and its *least significant* digit.

In this section we study the binary and ternary representations of natural numbers, instead of the decimal representation. The techniques and examples are, however, not specific for the binary or ternary number systems: they are applicable to any base.

We use finite lists of digits to represent numbers. For the sake of readability we use lists constructed by means of $\lhd$ – "snoc" –, instead of $\rhd$, but this is only notational: all solutions in terms of $\lhd$-lists can be transformed mechanically into isomorphic solutions in terms of $\rhd$-lists.

7.3.0 binary and ternary representations

We use $\mathcal{L}2$ as a shorthand for the type $\mathcal{L}_*(\{0,1\})$ of finite lists with elements in $\{0,1\}$ only. Also, we use $\mathcal{L}3$ as a shorthand for the type $\mathcal{L}_*(\{0,1,2\})$ of finite lists with elements in $\{0,1,2\}$ only. Lists in $\mathcal{L}2$ represent naturals according to the binary number system and lists in $\mathcal{L}3$ represent naturals according to the ternary number system. This is reflected by the following definitions of the abstraction functions $v2$ and $v3$, of types $\mathcal{L}2 \to \mathsf{Nat}$ and $\mathcal{L}3 \to \mathsf{Nat}$, respectively:

$$
\begin{array}{lcl}
v2 \cdot [\,] & = & 0 \\
v2 \cdot (s \lhd b) & = & 2 * v2 \cdot s + b \\
\\
v3 \cdot [\,] & = & 0 \\
v3 \cdot (s \lhd b) & = & 3 * v3 \cdot s + b
\end{array}
$$

So, according to this interpretation, the least significant digit of list $s \lhd b$ is b and s is the list remaining after removal of this least significant digit.

For example, both in binary and in ternary, $[\,]$ represents 0, and so does the list $[0]$. The binary list $[0,1,0,1]$ represents natural number 5, that

is: $v2 \cdot [0,1,0,1] = 5$. As a ternary list, the same list $[0,1,0,1]$ represents natural number 10, that is: $v3 \cdot [0,1,0,1] = 10$. This also shows that the number represented is not only determined by the list of digits itself, but also by the abstraction function used: the latter defines how the elements of the concrete data type are to be interpreted.

Notice that functions $v2$ and $v3$ are well-defined on any (finite) list of integers, even negative ones: the restriction of the elements to binary or ternary digits is only imposed for reasons of efficiency but is of no relevance to the meaning of $v2$ and $v3$.

7.3.1 computing a representation

The recursive definitions for $v2$ and $v3$ are admissible declarations in the functional language; as such, they can be used in programs, for instance to read sequences of digits from an input medium and calculate the numbers thus represented.

Conversely, we may be interested in a function $r3$, say, that maps natural numbers to suitable ternary representations. Such functions are useful to present natural number as sequences of digits to an output medium, like a modem or a printer. A suitable specification for $r3$ is:

(6) $\quad r3 \in \mathsf{Nat} \to \mathcal{L}3$, and:

(7) $\quad (\forall n :: v3 \cdot (r3 \cdot n) = n)$.

Requirement (6) states that $r3 \cdot n$ is a list of ternary digits and (7) states that $r3 \cdot n$ represents n; notice that this specification does not restrict $r3 \cdot n$ to the *shortest* possible list.

Because $v3 \cdot [n] = n$, for any natural n, the proposal $r3 \cdot n = [n]$ satisfies (7), but in general n is not a ternary digit and, hence, $[n]$ does not satisfy (6). So, in deriving a solution we must take into account both (6) and (7). We do so by taking (7) as the starting point for a derivation and by restricting ourselves to design decisions that lead to a solution satisfying (6) as well.

A general strategy to solve an equation of the shape $x : f \cdot x = E$ is to rewrite right-hand side expression E into an equivalent expression of the shape $f \cdot F$. This transforms the equation into the equivalent equation $x : f \cdot x = f \cdot F$, of which – thanks to Leibniz – expression F is an obvious solution.

In our example, we are heading for a declaration for $r3 \cdot n$ and the equation to be solved is $x : v3 \cdot x = n$. So, we rewrite right-hand side expression n into an equivalent expression of the shape $v3 \cdot F$. Because this inevitably will

involve the definition of $v3$ we follow the case distinction in that definition. Thus, we derive for the case $n=0$:

$$
\begin{array}{ll}
 & v3\cdot(r3\cdot 0) \\
= & \quad \{ \text{ specification (7) of } r3 \ \} \\
 & 0 \\
= & \quad \{ \text{ definition of } v3 \ \} \\
 & v3\cdot[\,] \quad ,
\end{array}
$$

from which we conclude that:

$$r3\cdot 0 \ = \ [\,]$$

satisfies (7); in addition this satisfies (6) because $[\,]$ is an element of $\mathcal{L}3$.

Furthermore, for natural $n, 1\leq n$, we derive:

$$
\begin{array}{ll}
 & v3\cdot(r3\cdot n) \\
= & \quad \{ \text{ specification (7) of } r3 \ \} \\
 & n \\
= & \quad \{ \text{ let } m \text{ and } b \text{ satisfy } n = 3*m+b \ \} \\
 & 3*m+b \\
= & \quad \{ \text{ specification (7) of } r3\text{, by Induction Hypothesis (see below) } \} \\
 & 3*v3\cdot(r3\cdot m)+b \\
= & \quad \{ \text{ definition of } v3 \text{, with } s := r3\cdot m \ \} \\
 & v3\cdot(r3\cdot m \triangleleft b) \quad ,
\end{array}
$$

from which we conclude that:

$$r3\cdot n \ = \ r3\cdot m \triangleleft b \quad ,$$

where $m, b\colon n = 3*m+b$, satisfies specification (7). In order that the appeal to the Induction Hypothesis is justified we need $m<n$. This follows from the assumption $1\leq n$ and from the additional requirement that b be a (nonnegative) natural number.

The purpose of the induction step is to introduce $v3$ into the formula; after all, we are heading for an expression of the shape $v3\cdot F$. Notice that the

second alternative of $v3$'s definition is only applicable to formulae in which $v3$ already occurs.

In order to satisfy specification (6) the elements of the list must be members of the set $\{0,1,2\}$. This is the case if b in $r3{\cdot}m \triangleleft b$ is in $\{0,1,2\}$. Thus, we obtain as complete specification for the numbers m, b:

$$n = 3*m+b \ \wedge \ 0 \leq b < 3 \quad ,$$

with as straightforward solution:

$$m = n \,\mathsf{div}\, 3 \ \wedge \ b = n \,\mathsf{mod}\, 3 \quad .$$

Combining the results we obtain this resursive declaration for function $r3$:

$$\begin{array}{lll} r3{\cdot}n & = & \mathsf{if} \ \ n=0 \ \rightarrow \ [\,] \\ & & [\!] \ \ n \geq 1 \ \rightarrow \ r3{\cdot}m \triangleleft b \\ & & \qquad\qquad\quad \mathsf{whr} \ \ m = n \,\mathsf{div}\, 3 \ \ \& \ \ b = n \,\mathsf{mod}\, 3 \ \ \mathsf{end} \\ & & \mathsf{fi} \end{array}$$

Defined this way, $r3{\cdot}n$ yields the shortest possible list of ternary digits representing n. In particular, 0 is represented by $[\,]$ and because $n \geq 1 \ \Rightarrow \ n \,\mathsf{div}\, 3 \neq 0 \ \vee \ n \,\mathsf{mod}\, 3 \neq 0$, the list $r3{\cdot}m \triangleleft b$ contains no leading zeroes. For shortest representations by *nonempty* lists we refer to the exercises.

7.3.2 addition of binary numbers

We consider a function add, of type $\mathcal{L}2 \rightarrow \mathcal{L}2 \rightarrow \mathcal{L}2$, and specified by:

(8) $\quad v2{\cdot}(add{\cdot}s{\cdot}t) \ = \ v2{\cdot}s + v2{\cdot}t \quad ,$ for all s, t of equal length .

In words, add maps binary lists –of the same length– to a binary list representing the sum of the numbers represented by its arguments. We derive a solution by induction on the lengths of the lists:

$$\begin{array}{ll} & v2{\cdot}(add{\cdot}[\,]{\cdot}[\,]) \\ = & \quad \{ \text{ specification (8) of } add \ \} \\ & v2{\cdot}[\,] + v2{\cdot}[\,] \\ = & \quad \{ \text{ definition of } v2 \text{ (twice) } \} \\ & 0 + 0 \\ = & \quad \{ \text{ algebra } \} \end{array}$$

$$
\begin{array}{ll}
 & 0 \\
= & \quad \{ \text{ definition of } v2 \ \} \\
 & v2 \cdot [\,] \quad ,
\end{array}
$$

from which we conclude that:

$$add \cdot [\,] \cdot [\,] \ = \ [\,]$$

satisfies (8) for the case that both s and t are empty. Because $[\,]$ has type $\mathcal{L}2$, this declaration also satisfies the type requirement for add.

So, what remains is the derivation of a solution for the case that both lists are non-empty:

$$
\begin{array}{ll}
 & v2 \cdot (\, add \cdot (s \triangleleft b) \cdot (t \triangleleft c)\,) \\
= & \quad \{ \text{ specification (8) of } add \ \} \\
 & v2 \cdot (s \triangleleft b) \ + \ v2 \cdot (t \triangleleft c) \\
= & \quad \{ \text{ definition of } v2 \text{ (twice) } \} \\
 & 2 * v2 \cdot s \ + \ b \ + \ 2 * v2 \cdot t \ + \ c \\
= & \quad \{ \text{ algebra } \} \\
 & 2 * (\, v2 \cdot s + v2 \cdot t\,) \ + \ b \ + \ c \\
= & \quad \{ \text{ specification (8) of } add\text{, by Induction Hypothesis } \} \\
 & 2 * v2 \cdot (\, add \cdot s \cdot t\,) \ + \ b \ + \ c \quad .
\end{array}
$$

Here we are in trouble: because all we know about b and c is $0 \leq b \leq 1$ and $0 \leq c \leq 1$, all we can conclude about $b+c$ is $0 \leq b+c \leq 2$. In order to be able to apply the definition of $v2$ –so as to move the occurrence of $v2$ to the top of the formula– and to satisfy the typing requirement, we would like $b+c$ to be a binary digit.

Therefore, we proceed with the derivation, as follows:

$$
\begin{array}{ll}
 & 2 * v2 \cdot (\, add \cdot s \cdot t\,) \ + \ b \ + \ c \\
= & \quad \{ \text{ algebra } \} \\
 & 2 * v2 \cdot (\, add \cdot s \cdot t\,) \ + \ 2 * (\,(b+c) \,\mathsf{div}\, 2\,) \ + \ (b+c) \,\mathsf{mod}\, 2 \\
= & \quad \{ \text{ algebra } \} \\
 & 2 * (\, v2 \cdot (\, add \cdot s \cdot t\,) + (b+c) \,\mathsf{div}\, 2\,) \ + \ (b+c) \,\mathsf{mod}\, 2 \quad .
\end{array}
$$

Now we are stuck again but for a different reason: the subexpression $(b+c)\,\mathsf{mod}\,2$ is a binary digit indeed, but now the whole expression does not match (the right-hand side of) $v2$'s definition anymore, because of the additional term $(b+c)\,\mathsf{div}\,2$.

To resolve this we generalize function add, by introducing an additional parameter that provides for the additional term. We call this function adc, and its specification becomes, for binary lists s, t and for d:

$$(9) \qquad v2\cdot(adc\cdot d\cdot s\cdot t) \;=\; v2\cdot s + v2\cdot t + d \quad , \text{ for } s,t \text{ of equal length} .$$

The new parameter d is the "carry" from a bit position into the next, more significant, bit position. As we will see below, its value is confined to the set $\{0,1\}$. Notice that the introduction of this carry parameter does require not any ingenuity at all: it just emerges as an unevitable generalization.

Function add can now be defined in terms of adc, because it is a special instance of it, with $d=0$:

$$add \;=\; adc\cdot 0 \quad .$$

To obtain a declaration for adc we redo the above derivations for add:

$$\begin{array}{ll}
 & v2\cdot(adc\cdot d\cdot[\,]\cdot[\,]) \\
= & \quad \{ \text{ specification (9) of } adc \text{ } \} \\
 & v2\cdot[\,] + v2\cdot[\,] + d \\
= & \quad \{ \text{ definition of } v2 \text{ (twice) } \} \\
 & 0 + 0 + d \\
= & \quad \{ \text{ algebra } \} \\
 & d \quad .
\end{array}$$

From this point onwards we can proceed in several, slightly different, ways. First, we can exploit that d is a binary digit and that, hence, it is adequately represented by the list $[d]$. This leads to a solution for adc where the function's value – a list – is exactly one element longer than the function's arguments. Second, we can distinguish the two cases $d=0$ and $d=1$, and represent 0 by $[\,]$ and represent 1 by $[1]$. One may be tempted to believe that this yields a solution with the shortest possible representation of the answer, but this is not true: if s and t both contain (redundant) leading zeroes then so may $adc\cdot d\cdot s\cdot t$.

Here we choose our first (and simplest) option, and define:

$$adc \cdot d \cdot [\,] \cdot [\,] \;=\; [\,d\,] \quad .$$

For the case that both lists are non-empty we derive:

$$\begin{array}{ll}
 & v2 \cdot (\, adc \cdot d \cdot (s \triangleleft b) \cdot (t \triangleleft c)\,) \\
= & \quad \{ \text{ specification (9) of } adc \text{ } \} \\
 & v2 \cdot (s \triangleleft b) \;+\; v2 \cdot (t \triangleleft c) \;+\; d \\
= & \quad \{ \text{ definition of } v2 \text{ (twice) } \} \\
 & 2 * v2 \cdot s \;+\; b \;+\; 2 * v2 \cdot t \;+\; c \;+\; d \\
= & \quad \{ \text{ algebra } \} \\
 & 2 * (v2 \cdot s + v2 \cdot t) \;+\; b \;+\; c \;+\; d \\
= & \quad \{ \text{ algebra, with } h = b+c+d \text{, as abbreviation } \} \\
 & 2 * (v2 \cdot s + v2 \cdot t) \;+\; 2 * (h \,\mathsf{div}\, 2) \;+\; h \,\mathsf{mod}\, 2 \\
= & \quad \{ \text{ algebra } \} \\
 & 2 * (v2 \cdot s + v2 \cdot t + h \,\mathsf{div}\, 2) \;+\; h \,\mathsf{mod}\, 2 \\
= & \quad \{ \text{ specification (9) of } adc\text{, by Induction Hypothesis } \} \\
 & 2 * v2 \cdot (\, adc \cdot (h \,\mathsf{div}\, 2) \cdot s \cdot t\,) \;+\; h \,\mathsf{mod}\, 2 \\
= & \quad \{ \text{ definition of } v2 \text{ } \} \\
 & v2 \cdot (\, adc \cdot (h \,\mathsf{div}\, 2) \cdot s \cdot t \;\triangleleft\; h \,\mathsf{mod}\, 2\,) \quad .
\end{array}$$

The appeal, by Induction Hypothesis, to adc's specification is justified, because $0 \le b \le 1$ and $0 \le c \le 1$ and $0 \le d \le 1$ together imply $0 \le h \le 3$ and, hence, $0 \le h \,\mathsf{div}\, 2 \le 1$. So, if d is in $\{0,1\}$ then $h \,\mathsf{div}\, 2$ is in $\{0,1\}$ as well.

Thus we obtain this declaration for function adc.

$$\begin{array}{lll}
 & adc \cdot d \cdot [\,] \cdot [\,] & = \; [\,d\,] \\
\& & adc \cdot d \cdot (s \triangleleft b) \cdot (t \triangleleft c) & = \; adc \cdot (h \,\mathsf{div}\, 2) \cdot s \cdot t \;\triangleleft\; h \,\mathsf{mod}\, 2 \\
 & & \quad \mathsf{whr} \;\; h \;=\; b+c+d \;\; \mathsf{end}
\end{array}$$

7.3.3 carry-save adders

The definition for function adc, as derived in the previous subsection, defines adc recursively in terms of adc itself and a pair of values $h \,\mathsf{div}\, 2$ and $h \,\mathsf{mod}\, 2$. When applied to lists of length n, the unfoldings of adc give rise to (exactly) n such pairs, and sequential computation of these pairs requires $\mathcal{O}(n)$ evaluation steps.

In addition, $h \,\mathsf{div}\, 2$ is the value for parameter d in the recursive application of adc, and this d occurs in the "next" two values $h \,\mathsf{div}\, 2$ and $h \,\mathsf{mod}\, 2$ again. As a result, each next such pair depends on the preceding pair; therefore, the required computation time would remain linear even when the n pairs $h \,\mathsf{div}\, 2$ and $h \,\mathsf{mod}\, 2$ would be evaluated concurrently!

aside: In combinatorial digital circuits subexpressions are evaluated concurrently, because all basic components of the circuit operate concurrently. A circuit that calculates $h \,\mathsf{div}\, 2$ and $h \,\mathsf{mod}\, 2$ from inputs b, c, and d is known as a *full adder* and a sequence of n such full adders – if connected properly – forms an n-bit adder; this circuit can be viewed as a direct implementation of function adc. The $\mathcal{O}(n)$ time complexity of adc's definition corresponds directly to the $\mathcal{O}(n)$ *propagation delay* of the circuit. As explained above, this is caused by the dependence of each full adder on its predecessor in the sequence, particularly, by the dependence of each subsequent carry on its preceding carry. For this reason this circuit is also known as a *ripple-carry adder*.

To obtain a design of a circuit with $\mathcal{O}(1)$ propagation delay the full adders must operate truly concurrently; therefore, we must uncouple the adders in such a way that each carry to the next stage does not depend anymore on the preceding stage. This is the motivation behind the following approach.

□

We introduce a function $adsc$, of type $\{0,1\} \to \mathcal{L}3 \to \mathcal{L}2 \to \mathcal{L}3$, with this specification, for binary d, ternary list s, and binary list s:

$$(10) \qquad v2 \cdot (adsc \cdot d \cdot s \cdot t) \;=\; v2 \cdot s + v2 \cdot t + d \quad , \text{ for } s, t \text{ of equal length .}$$

Notice that, as far as the numbers represented are concerned, $adsc$ has exactly the same specification as adc –see (9)–: both implement the same abstract function "addition with carry"; the only difference lies in *how* the numbers are represented.

As before, by the same derivation we obtain that

$$adsc \cdot d \cdot [\,] \cdot [\,] \;=\; [d]$$

satisfies (10); furthermore, for the case that both lists are non-empty we derive, in a way very similar to the derivation for adc:

$$
\begin{array}{ll}
 & v2\cdot(\,adsc\cdot d\cdot(s\triangleleft b)\cdot(t\triangleleft c)\,) \\
= & \quad \{\ \text{specification (10) of } adsc\ \} \\
 & v2\cdot(s\triangleleft b)\ +\ v2\cdot(t\triangleleft c)\ +\ d \\
= & \quad \{\ \text{definition of } v2 \text{ (twice)}\ \} \\
 & 2*v2\cdot s\ +\ b\ +\ 2*v2\cdot t\ +\ c\ +\ d \\
= & \quad \{\ \text{algebra}\ \} \\
 & 2*(v2\cdot s+v2\cdot t)\ +\ b\ +\ c\ +\ d \\
= & \quad \{\ \text{algebra, with } h=b+c\text{, as abbreviation}\ \} \\
 & 2*(v2\cdot s+v2\cdot t)\ +\ 2*(h\,\mathsf{div}\,2)\ +\ h\,\mathsf{mod}\,2\ +\ d \\
= & \quad \{\ \text{algebra}\ \} \\
 & 2*(v2\cdot s+v2\cdot t+h\,\mathsf{div}\,2)\ +\ h\,\mathsf{mod}\,2\ +\ d \\
= & \quad \{\ \text{specification (10) of } adsc\text{, by Induction Hypothesis}\ \} \\
 & 2*v2\cdot(\,adsc\cdot(h\,\mathsf{div}\,2)\cdot s\cdot t\,)\ +\ h\,\mathsf{mod}\,2\ +\ d \\
= & \quad \{\ \text{definition of } v2\ \} \\
 & v2\cdot(\,adsc\cdot(h\,\mathsf{div}\,2)\cdot s\cdot t\ \triangleleft\ (\,h\,\mathsf{mod}\,2+d\,)\,)\quad ,
\end{array}
$$

as a result of which we obtain this declaration for $adsc$:

$$
\begin{array}{lll}
 & adsc\cdot d\cdot[\,]\cdot[\,] & = & [\,d\,] \\
\& & adsc\cdot d\cdot(s\triangleleft b)\cdot(t\triangleleft c) & = & adsc\cdot(h\,\mathsf{div}\,2)\cdot s\cdot t\ \triangleleft\ (\,h\,\mathsf{mod}\,2+d\,) \\
 & & & & \mathsf{whr}\ \ h\ =\ b+c\ \ \mathsf{end}
\end{array}
$$

All type requirements are met: $b\leq 2$ and $c\leq 1$ imply $h\leq 3$, hence, $h\,\mathsf{div}\,2\leq 1$. Also, if in addition $d\leq 1$ then $h\,\mathsf{mod}\,2+d\leq 2$ as well.

In this definition the value, $h\,\mathsf{div}\,2$, for parameter d in the recursive application of $adsc$ now depends on b and c only, and not on d; as a result, this definition allows truly concurrent evaluation of all pairs $h\,\mathsf{div}\,2$ and $h\,\mathsf{mod}\,2+d$.

aside: As a result, function $adsc$ can be implemented as a combinatorial digital circuit with constant propagation delay, that is, independent of the length of the lists. Such a circuit is known as a *carry-save* adder.

The gain in efficiency is obtained by admitting redundancy in the representation of the numbers: the use of $\{0,1,2\}$ instead of $\{0,1\}$ for the digits of the lists, with the *same* abstraction function $v2$, provides the flexibility that makes this more efficient solution possible.

Because $\{0,1\} \subseteq \{0,1,2\}$ we also have $\mathcal{L}2 \subseteq \mathcal{L}3$: that *adsc* 's parameter s must have type $\mathcal{L}3$ (instead of $\mathcal{L}2$) is not a restriction but a relaxation. Hence, *adsc* can be applied safely to binary lists. Conversely, that *adsc* 's value has type $\mathcal{L}3$ (instead of $\mathcal{L}2$) means that its value is not guaranteed to be in (truly) binary representation.

To convert the value of *adsc* to a binary list an additional function is needed – see the exercises –, the circuit implementation of which happens to have a linear propagation delay again. This is not so strange: the final carry propagation has to occur somewhere. So, what have we gained then? If *adsc* would be used for a single addition nothing would be gained at all. If, however, the function is used (repeatedly) to add a collection of $m+1$ numbers then the m additions require no more than $\mathcal{O}(m)$ time plus once the time needed to convert the final result to binary. Hence, with carry-save adders the addition of $m+1$ numbers in n bit binary representation takes $\mathcal{O}(m)+\mathcal{O}(n)$ time instead of $\mathcal{O}(m)*\mathcal{O}(n)$, as would be needed with ordinary adders.

□

7.3.4 representation conversions

An interesting problem – and for many a programmer a tough one – is the conversion of the representation of a number in one base to the representation of that same number in another base. Here we confine ourselves to the conversion from binary lists to ternary lists. The reverse problem, from ternary to binary, is slightly more complicated; this is left to the exercises.

The conversion problem is simple to specify: we are looking for a function *c23*, of type $\mathcal{L}2 \to \mathcal{L}3$, that satisfies:

(11) $\quad v3\cdot(c23\cdot s) \;=\; v2\cdot s$, for all s in $\mathcal{L}2$.

A very straightforward solution is to compose function *r3* – from Section 7.3.1 – with *v2*:

$$
\begin{array}{ll}
 & v3\cdot(c23\cdot s) \\
= & \quad \{ \text{ specification (11) of } c23 \text{ } \} \\
 & v2\cdot s \\
= & \quad \{ \text{ specification (7) of } r3 \text{, with } n := v2\cdot s \text{, to reintroduce } v3 \text{ } \} \\
 & v3\cdot(r3\cdot(v2\cdot s)) \quad ,
\end{array}
$$

Thus we obtain:

(12) $\quad c23 \cdot s \;=\; r3 \cdot (v2 \cdot s) \quad .$

This solution is correct, but the (maximal) value of the integer subexpression $v2 \cdot s$ increases exponentially with the length of list s. When the program is supposed to be executed by a machine with a limited range of integer values, this exponential behaviour may become prohibitive, particularly if the function is applied to (very) long lists.

We now develop two solutions that satisfy the additional requirement that all integer subexpressions have bounded values. Here "bounded" means: independent of the length of the list. As a matter of fact, our solutions will be such that all integer subexpressions have values in a rather small range.

7.3.5 by repeated multiplication

Using induction on s we derive:

$$
\begin{array}{ll}
 & v3 \cdot (c23 \cdot [\,]) \\
= & \quad \{ \text{ specification (11) of } c23 \text{ } \} \\
 & v2 \cdot [\,] \\
= & \quad \{ \text{ definition of } v2 \text{ , to eliminate } v2 \text{ } \} \\
 & 0 \\
= & \quad \{ \text{ definition of } v3 \text{ , to introduce } v3 \text{ } \} \\
 & v3 \cdot [\,] \quad .
\end{array}
$$

So, for the empty list we obtain as a solution:

$$c23 \cdot [\,] \;=\; [\,] \quad .$$

Furthermore, we derive:

$$
\begin{array}{ll}
 & v3 \cdot (c23 \cdot (s \triangleleft b)) \\
= & \quad \{ \text{ specification (11) of } c23 \text{ } \} \\
 & v2 \cdot (s \triangleleft b) \\
= & \quad \{ \text{ definition of } v2 \text{ } \} \\
 & 2 * v2 \cdot s + b \\
= & \quad \{ \text{ specification (11) of } c23\text{, by Induction Hypothesis } \}
\end{array}
$$

$$
\begin{array}{ll}
 & 2 * v3 \cdot (c23 \cdot s) + b \\
= & \quad \{ \text{ introduction of function } f \text{ (see below) } \} \\
 & v3 \cdot (f \cdot b \cdot (c23 \cdot s)) \quad .
\end{array}
$$

The purpose of the induction step is to introduce $v3$ into the formula and to eliminate $v2$. The purpose of function f is to move the occurrence of $v3$ to the top of the formula, as needed to solve the equation. We really need a new function here because the shape of the formula $2 * v3 \cdot (c23 \cdot s) + b$ does not match anything already available.

Combining the results of the two derivations we obtain as declaration for $c23$, in terms of our new function f:

$$
\begin{array}{lll}
 & c23 \cdot [\,] & = & [\,] \\
\& & c23 \cdot (s \triangleleft b) & = & f \cdot b \cdot (c23 \cdot s)
\end{array}
$$

Function f maps a binary digit b and a ternary list t to a ternary list, according to this specification:

$$
(13) \qquad v3 \cdot (f \cdot b \cdot t) \;=\; 2 * v3 \cdot t + b \quad .
$$

In words, f implements the operation mapping binary digit b and natural n to $2 * n + b$, where both n and the result are represented in ternary. This is the core operation in the definition of the abstraction function $v2$ and the above definition for $c23$ effectively implements $v2$ in the ternary system.

We are left with the obligation to derive a declaration for f. We do so by induction on t:

$$
\begin{array}{ll}
 & v3 \cdot (f \cdot b \cdot [\,]) \\
= & \quad \{ \text{ specification (13) of } f \text{ } \} \\
 & 2 * v3 \cdot [\,] + b \\
= & \quad \{ \text{ definition of } v3 \text{, algebra } \} \\
 & b \\
= & \quad \{ \text{ algebra, definition of } v3 \text{ } \} \\
 & 3 * v3 \cdot [\,] + b \\
= & \quad \{ \text{ definition of } v3 \text{, and } [\,] \triangleleft b = [b] \text{ } \} \\
 & v3 \cdot [b] \quad .
\end{array}
$$

and:

$$
\begin{array}{ll}
 & v3\cdot(f\cdot b\cdot(t\lhd c)) \\
= & \quad \{ \text{ specification (13) of } f \text{ } \} \\
 & 2*v3\cdot(t\lhd c) + b \\
= & \quad \{ \text{ definition of } v3 \text{ } \} \\
 & 2*(3*v3\cdot t+c) + b \\
= & \quad \{ \text{ algebra } \} \\
 & 3*(2*v3\cdot t) + 2*c + b \\
= & \quad \{ \text{ algebra, with } h=2*c+b \text{ } \} \\
 & 3*(2*v3\cdot t+h\,\mathsf{div}\,3) + h\,\mathsf{mod}\,3 \\
= & \quad \{ \text{ specification (13) of } f \text{, by Induction Hypothesis } \} \\
 & 3*v3\cdot(f\cdot(h\,\mathsf{div}\,3)\cdot t) + h\,\mathsf{mod}\,3 \\
= & \quad \{ \text{ definition of } v3 \text{ } \} \\
 & v3\cdot(f\cdot(h\,\mathsf{div}\,3)\cdot t \lhd h\,\mathsf{mod}\,3) \quad .
\end{array}
$$

Because $0\le b\le 1$ is a precondition of $f\cdot b\cdot t$, the recursive application above has $0\le h\,\mathsf{div}\,3\le 1$ as a precondition. This condition is met because $0\le b\le 1$ and $0\le c\le 2$ imply $0\le h\le 5$ and, hence, $0\le h\,\mathsf{div}\,3\le 1$. This also shows that our wish to keep the values of all integer subexpressions bounded is fulfilled: the maximal value of h is 5.

Thus we obtain this declaration for function f.

$$
\begin{array}{lll}
 & f\cdot b\cdot[\,] & = & [b] \\
\& & f\cdot b\cdot(t\lhd c) & = & f\cdot(h\,\mathsf{div}\,3)\cdot t \lhd h\,\mathsf{mod}\,3 \\
 & & & \mathsf{whr}\ \ h = 2*c+b\ \ \mathsf{end}
\end{array}
$$

7.3.6 by repeated division

The previous solution boils down to an implementation of function $v2$ in the ternary number representation. Another solution is obtained by taking definition (12) for $c23$ as starting point:

(12) $\quad c23\cdot s \;=\; r3\cdot(v2\cdot s) \quad .$

Earlier we have rejected this definition because we decided to construct solutions in which all integer subexpressions have bounded values: in (12) the value of subexpression $v2\cdot s$ is not bounded.

We can remedy this, however, by implementing the operations on natural numbers, as they occur in (12), as operations in the binary number system. The reason to use binary representation is that $c23$'s parameter s is a binary list to start with. That is, we now use the definition for $r3$ to develop a declaration for the composition $r3 \circ v2$. Obviously, this will involve the definition of $v2$ as well.

In an approach like this we say that (the definitions of) $r3$ and $v2$ are *fused* into a single definition for their composition. Fusion is an instance of a technique for *program transformations*, which we will discuss more extensively in Chapter 8.

We recall $r3$'s definition (from Section 7.3.1):

$$
\begin{array}{lll}
r3 \cdot n & = & \mathsf{if}\ \ n = 0 \ \rightarrow\ [\,] \\
 & & [\!]\ \ n \geq 1 \ \rightarrow\ r3 \cdot m \lhd b \\
 & & \qquad\qquad\quad \mathsf{whr}\ \ m = n \,\mathsf{div}\, 3\ \ \&\ \ b = n \,\mathsf{mod}\, 3\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}
$$

We transform this into a definition for the composition $r3 \circ v2$, as follows. First, for s of type $\mathcal{L}2$, we substitute $v2 \cdot s$ for n:

$$
\begin{array}{lll}
r3 \cdot (v2 \cdot s) & = & \mathsf{if}\ \ v2 \cdot s = 0 \ \rightarrow\ [\,] \\
 & & [\!]\ \ v2 \cdot s \geq 1 \ \rightarrow\ r3 \cdot m \lhd b \\
 & & \qquad\qquad\quad \mathsf{whr}\ \ m = (v2 \cdot s) \,\mathsf{div}\, 3 \\
 & & \qquad\qquad\quad\ \ \&\ \ b = (v2 \cdot s) \,\mathsf{mod}\, 3\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}
$$

Second, because we wish to replace the recursive application of $r3$ by a recursive application of $r3 \circ v2$, we introduce a new variable t, of type $\mathcal{L}2$ and which will represent m (in $r3 \cdot m$); we substitute $v2 \cdot t$ for m and we replace the declaration of m (in the where-clause) by a specification for our new variable t:

$$
\begin{array}{lll}
r3 \cdot (v2 \cdot s) & = & \mathsf{if}\ \ v2 \cdot s = 0 \ \rightarrow\ [\,] \\
 & & [\!]\ \ v2 \cdot s \geq 1 \ \rightarrow\ r3 \cdot (v2 \cdot t) \lhd b \\
 & & \qquad\qquad\quad \mathsf{whr}\ \ t : v2 \cdot t = (v2 \cdot s) \,\mathsf{div}\, 3 \\
 & & \qquad\qquad\quad\ \ \&\ \ b = (v2 \cdot s) \,\mathsf{mod}\, 3\ \ \mathsf{end} \\
 & & \mathsf{fi}
\end{array}
$$

To transform this into a useful recursive declaration we must implement the guards $v2 \cdot s = 0$ and $v2 \cdot s \geq 1$, and we must derive declarations for the local variables t and b.

The guards need attention because the subexpression $v2 \cdot s$ does not satisfy our requirement that all integer subexpressions in our solution have bounded values. In addition, the evaluation of these guards may take an amount of time proportional to the length of the list: if s contains zeroes only we have $v2 \cdot s = 0$ but a traversal of the whole list is needed to calculate this.

The first guard, $v2 \cdot s = 0$, may be safely *strengthened* to $s = [\,]$. Also, we *weaken* the other guard, $v2 \cdot s \geq 1$, to $s \neq [\,]$. Generally, weakening a guard is not safe, but in our case we recall that we are not really interested in function $r3$ itself: we are only using it to construct a declaration for $c23$, and $c23$'s specification (11) prescribes that:

$$v3 \cdot (c23 \cdot s) \;=\; v2 \cdot s \quad \text{, for all } s \text{ in } \mathcal{L}2 \quad ,$$

which is satified if we define $c23$ by:

$$c23 \cdot s \;=\; r3 \cdot (v2 \cdot s) \quad ,$$

but $c23$'s specification is also satisfied by the (weaker):

$$v3 \cdot (c23 \cdot s) \;=\; v3 \cdot (r3 \cdot (v2 \cdot s)) \quad .$$

As a consequence, we are not really interested in $r3$, we are only interested in $v3 \circ r3$. Now, $r3$ does not satisfy $r3 \cdot n \;=\; r3 \cdot m \triangleleft b$ –with a proper choice for m and b–, for all n, but it does satisfy $v3 \cdot (r3 \cdot n) \;=\; v3 \cdot (r3 \cdot m \triangleleft b)$, for all n. That is sufficient.

Finally, we are left with the problem to construct declarations for the local variables t and b. Because the calculation of (a list representing) $(v2 \cdot s)\,\mathsf{div}\,3$ unevitably involves $(v2 \cdot s)\,\mathsf{mod}\,3$ too, we may as well pair the definitions of t and b right away. Thus, we introduce a function g, of type $\mathcal{L}2 \rightarrow \langle \mathcal{L}2, \{0,1\} \rangle$, with this specification, for s in $\mathcal{L}2$:

$$(14) \qquad g \cdot s \;=\; \langle t, c \rangle \;\; \mathsf{whr} \;\; t : v2 \cdot t = (v2 \cdot s)\,\mathsf{div}\,3 \;\; \& \;\; c = (v2 \cdot s)\,\mathsf{mod}\,3 \;\; \mathsf{end}$$

Because $v2 \cdot [\,] = 0$ and because both $0\,\mathsf{div}\,3$ and $0\,\mathsf{mod}\,3$ are zero, the base case for a declaration for g becomes:

$$g \cdot [\,] \;=\; \langle [\,], 0 \rangle \quad ,$$

and for non-empty list $s \triangleleft b$ we derive:

$$
\begin{array}{ll}
 & (v2\cdot(s\triangleleft b))\ \mathsf{div}\ 3 \\
= & \quad \{\ \text{definition of } v2\ \} \\
 & (2*v2\cdot s+b)\ \mathsf{div}\ 3 \\
= & \quad \{\ \text{assume: } v2\cdot s\ =\ 3*v2\cdot t+c\,,\ \text{see below}\ \} \\
 & (2*(3*v2\cdot t+c)+b)\ \mathsf{div}\ 3 \\
= & \quad \{\ \text{algebra, with } h=2*c+b\ \} \\
 & 2*v2\cdot t\ +\ h\,\mathsf{div}\,3 \\
= & \quad \{\ \text{definition of } v2\ \} \\
 & v2\cdot(t\triangleleft h\,\mathsf{div}\,3)\quad .
\end{array}
$$

The assumption $v2\cdot s\ =\ 3*v2\cdot t+c$ can be met by a recursive application of g, by induction hypothesis, namely by defining t and c by:

$$\langle t,c\rangle\ =\ g\cdot s\quad .$$

The list $t\triangleleft h\,\mathsf{div}\,3$ is a binary list because, by induction hypothesis, t is in $\mathcal{L}2$ and because h is in the range $[\,0\,..\,5\,]$: this follows from $b\in\{0,1\}$ and from $c\in\{0,1,2\}$, which follows from g's specification.

Furthermore, we derive:

$$
\begin{array}{ll}
 & (v2\cdot(s\triangleleft b))\ \mathsf{mod}\ 3 \\
= & \quad \{\ \text{definition of } v2\ \} \\
 & (2*v2\cdot s+b)\ \mathsf{mod}\ 3 \\
= & \quad \{\ \text{assume: } v2\cdot s\ =\ 3*v2\cdot t+c\,,\ \text{as before}\ \} \\
 & (2*(3*v2\cdot t+c)+b)\ \mathsf{mod}\ 3 \\
= & \quad \{\ \text{algebra, with } h=2*c+b\ \} \\
 & h\,\mathsf{mod}\,3\quad .
\end{array}
$$

By combining the results of the latter two derivations, together with the base case, we obtain as recursive declaration for function g:

$$
\begin{array}{lll}
 & g\cdot[\,] & =\ \langle[\,],0\rangle \\
\& & g\cdot(s\triangleleft b) & =\ \langle t\triangleleft h\,\mathsf{div}\,3\,,\,h\,\mathsf{mod}\,3\rangle \\
 & & \quad \mathsf{whr}\ \ h\ =\ 2*c+b\ \ \&\ \ \langle t,c\rangle\ =\ g\cdot s\ \ \mathsf{end}
\end{array}
$$

We now use function g to construct a declaration for $c23$, as follows:

$$
\begin{array}{lll}
 & c23\cdot[\,] & = & [\,] \\
\& & c23\cdot(s\triangleleft b) & = & c23\cdot t \triangleleft c \\
 & & & \mathsf{whr}\ \langle t, c\rangle = g\cdot(s\triangleleft b)\ \ \mathsf{end}
\end{array}
$$

7.4 Exponentiation Revisited

In Section 3.4 we have studied the following recurrence relations for exponentiation, for integer x and natural n:

$$
\begin{array}{lll}
x^0 & = & 1 \\
x^{n+1} & = & x^n * x \\
x^{2*n} & = & (x^2)^n
\end{array}
$$

As we have seen, the last one of these relations allows calculation of x^n in $\mathcal{O}(^2log\cdot n)$ steps. The only algebraic properties of binary operator $*$ that are relevant for these recurrences are that $*$ has an identity element, namely 1, and that $*$ is associative. Therefore, the notion of exponentiation and these recurrence relations are meaningful for *any* associative binary operator with an identity element.

7.4.0 function composition

Function composition is associative and has the identity function I as its identity element. Hence, it is meaningful to define exponentation for functions, which amounts to n-fold composition of a function with itself, denoted by f^n, for any function f and natural n. The above recurrences can then be reformulated directly, for function f and natural n:

$$
\begin{array}{lll}
f^0 & = & I \\
f^{n+1} & = & f^n \circ f \\
f^{2*n} & = & (f^2)^n
\end{array}
$$

As before, the last of these relations allows calculation of f^n in $\mathcal{O}(^2log\cdot n)$ steps; without further provisions, however, this result is rather useless, because *application* of function f^n to an argument still requires n evaluation steps. For example, $f^2\cdot x = f\cdot(f\cdot x)$ and the time needed to evaluate $f\cdot(f\cdot x)$ is independent of how f^2 has been formed: function f must be applied twice anyhow.

7.4.1 linear combinations and matrices

We now consider the special case that f belongs to a class of functions that can be *represented* efficiently. We assume that this class is *closed under function composition*, which means that compositions of functions from the class belong to the class too and, hence, can be represented efficiently as well. In particular, if f is a function in this class then so is f^2. If the functions are represented in such a way that application to an argument of a function is an efficient operation then some gain in efficiency is possible: evaluation of $f^n \cdot x$ may then be equally efficient as evaluation of $f \cdot x$, provided f^n itself is available already. Now efficient exponentiation becomes useful.

A particularly interesting example is the class of *linear mappings* from a p-dimensional vector space to itself. A function from p-tuples to p-tuples is a linear mapping if every element of the function's value is a *linear combination* of the elements of the function's argument.

Linear mappings from a p-dimensional vector space to itself can be conveniently represented by $p \times p$ matrices: the matrix representing a function contains the coefficients for the linear combinations that make up the function's value. In this representation, function application amounts to matrix times vector multiplication, and function composition amounts to matrix times matrix multiplication. For example, with $\times$ for matrix multiplication, if matrix A represents function f then $A \times A$, or: A^2, represents f^2 and $A \times x$ represents $f \cdot x$, for vector x.

7.4.2 two examples

In Section 2.6.1 we have seen a function h mapping the naturals to pairs of successive Fibonacci numbers; that is, function h satisfies:

$$h \cdot n \;=\; \langle\, \mathit{fib} \cdot n \,,\, \mathit{fib} \cdot (n{+}1) \,\rangle \quad \text{, for natural } n \quad .$$

A recursive declaration for h is:

$$\begin{array}{lll} & h \cdot 0 & = & \langle 0, 1 \rangle \\ \& & h \cdot (n{+}1) & = & \langle y, x{+}y \rangle \;\; \mathsf{whr}\; \langle x, y \rangle = h \cdot n \;\; \mathsf{end} \end{array}$$

The pair $\langle y, x{+}y \rangle$ is a linear combination of its constituents x and y, which are the elements of $h \cdot n$. Hence, the pair $\langle y, x{+}y \rangle$ can be seen as the value of a linear mapping applied to $h \cdot n$. Calling this linear mapping k we have, for all x, y:

$$k \cdot \langle x, y \rangle \;=\; \langle y, x{+}y \rangle \quad ,$$

and in terms of k the definition for h can be rewritten as:

$$\begin{array}{lll} & h\cdot 0 & = & \langle 0,1\rangle \\ \& & h\cdot(n{+}1) & = & k\cdot(h\cdot n) \end{array}$$

Because k does not depend on n we obtain, by straightforward mathematical induction (and by using function composition):

$$h\cdot n \;=\; k^n\cdot(h\cdot 0) \quad \text{, for natural } n \quad ,$$

which is equivalent to:

$$(15) \qquad h\cdot n \;=\; k^n\cdot\langle 0,1\rangle \quad \text{, for natural } n \quad .$$

This provides the key to a more efficient implementation: if we succeed in implementing $k^n\cdot\langle 0,1\rangle$ with time complexity $\mathcal{O}(^2log\cdot n)$ then we also have a program for $fib\cdot n$ with this time complexity.

Function k is a linear mapping from pairs to pairs, which can be represented by the $2{\times}2$ matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, because $\begin{pmatrix} y \\ x{+}y \end{pmatrix}$ equals $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix}$.

We now generalize (15) by replacing constants k and $\langle 0,1\rangle$ by paramaters, of type matrix and vector, respectively. In the functional notation we can simply represent a matrix as a quadruple, and a vector can be represented as a pair: $\langle a,b,c,d\rangle$ and $\langle x,y\rangle$ thus represent $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and $\begin{pmatrix} x \\ y \end{pmatrix}$.

Using, for the time being, a binary operator $\otimes$ to represent both matrix by matrix and matrix by vector multiplication in terms of these quadruples and pairs, we introduce a new function gh, with this specification, for quadruple $-$matrix$-$ m, and pair $-$vector$-$ v, and natural n:

$$gh\cdot m\cdot v\cdot n \;=\; m^n \otimes v \quad .$$

As gh is a generalization of h we obtain that:

$$(16) \qquad h\cdot n \;=\; gh\cdot\langle 0,1,1,1\rangle\cdot\langle 0,1\rangle\cdot n \quad ,$$

then satisfies (15).

Using the known recurrence relations for exponentiation, we now obtain, without further difficulties, this recursive declaration for gh:

$$\begin{array}{lll} & gh\cdot m\cdot v\cdot 0 & = & v \\ \& & gh\cdot m\cdot v\cdot(n{+}1) & = & gh\cdot m\cdot(m\otimes v)\cdot n \\ \& & gh\cdot m\cdot v\cdot(2{*}n) & = & gh\cdot(m\otimes m)\cdot v\cdot n \end{array}$$

So as to really achieve the desired efficiency, and to make the recursion well-founded – recall that the last alternative is only meaningful if $1 \leq n$, see Section 3.4 –, we have to strengthen the guards somewhat. This yields as final declaration for gh:

$$\begin{array}{lll} & gh\cdot m\cdot v\cdot 0 & = & v \\ \& & gh\cdot m\cdot v\cdot(2{*}n{+}1) & = & gh\cdot m\cdot(m\otimes v)\cdot(2{*}n) \\ \& & gh\cdot m\cdot v\cdot(2{*}n{+}2) & = & gh\cdot(m\otimes m)\cdot v\cdot(n{+}1) \end{array}$$

The encoding of the matrix/vector multiplication $\otimes$ in terms of quadruples and pairs is left as an exercise.

remark: It is not really necessary to replace constant $\langle 0,1\rangle$ by a parameter. We could also have introduced a new function to implement k^n separately from the application to $\langle 0,1\rangle$. The version presented here is slightly simpler and slightly more efficient, because we exploit the definition of function composition: $(f^n\circ f)\cdot x$ is equal to $f^n\cdot(f\cdot x)$. Thus, one function composition is reduced right away to a function application.

In terms of matrix and vector multiplication this amounts to exploitation of the equality of $(m^n\otimes m)\otimes v$ and $m^n\otimes(m\otimes v)$.

□

* * *

In Section 3.8 we have, as one of the solutions for $(\Sigma i: 0\leq i<N: X^i)$, derived the following declaration for a function h, obtained by tupling two functions:

$$\begin{array}{lll} & h\cdot 0 & = & \langle 0,1\rangle \\ \& & h\cdot(n{+}1) & = & \langle\, x{+}y\,,\, X{*}y\,\rangle \;\; \textsf{whr}\; \langle x,y\rangle = h\cdot n \;\; \textsf{end} \end{array}$$

In terms of an additional function k, defined by – formula (20) in Section 3.8 –:

$$k\cdot\langle x,y\rangle \;=\; \langle\, x{+}y\,,\, X{*}y\,\rangle \quad ,$$

the second alternative of h's definition can be rewritten as:

$$h\cdot(n{+}1) \;=\; k\cdot(h\cdot n) \quad .$$

Because k does not depend on n we obtain, by straightforward mathematical induction (and by using function composition):

$$h{\cdot}n \;=\; k^n{\cdot}(h{\cdot}0) \quad \text{, for natural } n \quad ,$$

which is equivalent to:

$$h{\cdot}n \;=\; k^n{\cdot}\langle 0,1\rangle \quad \text{, for natural } n \quad .$$

As in the previous example, function k is a linear mapping from pairs to pairs and k is represented by the $2{\times}2$ matrix $\begin{pmatrix} 1 & 1 \\ 0 & X \end{pmatrix}$.

Notice that, apart from function k, this example now has exactly the same structure as the previous one: the same function gh can be used to define our current h, we only have to substitute a different matrix. Our current h can now be defined thus:

$$h{\cdot}n \;=\; gh{\cdot}\langle 1,1,0,X\rangle{\cdot}\langle 0,1\rangle{\cdot}n \quad .$$

This way we obtain yet another $\mathcal{O}({}^2log{\cdot}N)$ solution for the computation of $(\Sigma i : 0{\le}i{<}N : X^i)$.

7.4.3 embellishments

In the Fibonacci example function gh is applied to the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ (in formula (16)). Now, we have that $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2$ equals $\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$ and $\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}^2$ equals $\begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}$, and so on. Apparently, it is a precondition of (this particular use of) function gh that its parameter m invariantly has the general shape $\begin{pmatrix} a & b \\ b & a{+}b \end{pmatrix}$, for some natural a and b: if m has this shape then so has m^2. Hence, matrix m can be represented by just two naturals a and b and instead of a quadruple we only need a pair of naturals now, or just two natural parameters instead of a pair.

This idea can implemented as follows, where, just for the sake of illustration, we use simple natural parameters instead of pairs: parameters a and b represent matrix m and parameters x and y represent pair v. Calling this (version of the) function $gh1$, all its parameters have type Nat now and its specification becomes:

$$gh1{\cdot}a{\cdot}b{\cdot}x{\cdot}y{\cdot}n \;=\; gh{\cdot}\langle a,b,b,a{+}b\rangle{\cdot}\langle x,y\rangle{\cdot}n \quad .$$

Using this specification and the recursive declaration for gh a declaration for $gh1$ is easily constructed:

$$
\begin{array}{lll}
 & gh1\cdot a\cdot b\cdot x\cdot y\cdot 0 & = & \langle x,y\rangle \\
\& & gh1\cdot a\cdot b\cdot x\cdot y\cdot (2{*}n{+}1) & = & gh1\cdot a\cdot b\cdot (a{*}x{+}b{*}y)\cdot (b{*}x{+}(a{+}b){*}y)\cdot (2{*}n) \\
\& & gh1\cdot a\cdot b\cdot x\cdot y\cdot (2{*}n{+}2) & = & gh1\cdot (a^2{+}b^2)\cdot (2{*}a{*}b{+}b^2)\cdot x\cdot y\cdot (n{+}1)
\end{array}
$$

The transition from gh to $gh1$ is an example of a *program transformation*, to which we will devote an entire chapter.

* * *

A similar encoding trick can be applied to the other example: the initial matrix is $\begin{pmatrix} 1 & 1 \\ 0 & X \end{pmatrix}$ and its square is: $\begin{pmatrix} 1 & 1{+}X \\ 0 & X^2 \end{pmatrix}$. The general (invariant) shape of this matrix is $\begin{pmatrix} 1 & a \\ 0 & b \end{pmatrix}$, because its square equals $\begin{pmatrix} 1 & 1{+}a{*}b \\ 0 & b^2 \end{pmatrix}$.

Again, this can be exploited by representing the matrix by just two variables a and b. The further elaboration of this is as easy as in the previous example.

7.5 Exercises

0. Prove that leading zeroes do not affect the number represented by a digit string; that is, prove (in the ternary system): $v3\cdot([0] +\!\!+ s) = v3\cdot s$.

1. Derive a declaration for function $r3$, with the same specification as in Section 7.3.1, but with the additional requirement that $r3\cdot n$ is the *shortest non-empty* ternary list representing natural number n. In particular, this implies that $r3\cdot 0 = [0]$.

2. (a) Formulate a recursive definition for a predicate *shortest* on $\mathcal{L}3$, that distinguishes the shortest ternary representations from the other ones.
   (b) Derive a declaration for a function *trunc* that maps a ternary list onto the equivalent shortest ternary list; here "equivalent" means: representing the same natural number.

3. Derive declarations for functions *inc*, *dub*, *trp*, *hlv*, and *trd*, each of them of type $\mathcal{L}2 \to \mathcal{L}2$, with these specifications, for all s in $\mathcal{L}2$:

$$
\begin{array}{lll}
v2\cdot(inc\cdot s) & = & 1 + v2\cdot s \\
v2\cdot(dub\cdot s) & = & 2 * v2\cdot s \\
v2\cdot(trp\cdot s) & = & 3 * v2\cdot s \\
v2\cdot(hlv\cdot s) & = & v2\cdot s \ \mathsf{div}\ 2 \\
v2\cdot(trd\cdot s) & = & v2\cdot s \ \mathsf{div}\ 3
\end{array}
$$

4. Derive a declaration for a function add, of type $\mathcal{L}2 \to \mathcal{L}2 \to \mathcal{L}2$, and specified by: $v2 \cdot (add \cdot s \cdot t) = v2 \cdot s + v2 \cdot t$, for all s, t, not necessarily of equal length, in $\mathcal{L}2$.

5. Derive a declaration for a function $c32$ of type $\mathcal{L}3 \to \mathcal{L}2$ and such that $(\forall s :: v2 \cdot (c32 \cdot s) = v3 \cdot s)$; as in Section 7.3.4 all integer subexpressions must have bounded values.

6. In the, so-called, *two's complement* representation, lists of binary digits are used to represent (both positive and negative) integers. As in the ordinary binary system, the representing data type is $\mathcal{L}2$, but the empty list is not used and a different abstraction function is used, called $vn2$ here and defined recursively by:

$$\begin{array}{lcl} vn2 \cdot [b] & = & -b \\ vn2 \cdot (s \triangleleft b) & = & 2 * vn2 \cdot s + b \end{array}$$

   (a) Prove that duplication of the leading digit does not affect the number represented by a digit string; that is, for any $d \in \{0, 1\}$ and any $s \in \mathcal{L}2$, prove: $vn2 \cdot ([d, d] +\!\!+ s) = vn2 \cdot ([d] +\!\!+ s)$.

   (b) Prove that every integer is representable, by deriving a declaration for a function $rn2$, of type $\mathsf{Int} \to \mathcal{L}2$, that maps every integer onto a two's complement representation.

   (c) As exercise 3, but now with $v2$ replaced by $vn2$.

   (d) Specify and derive a declaration for a function add that implements addition of integers in two's complement representation. You may assume that add's parameters have equal lengths.

   (e) Similarly, specify and derive a declaration for a function $subt$ that implements subtraction of integers in two's complement representation. You may assume that $subt$'s parameters have equal lengths.

7. Derive a declaration for a function adj, of type $\mathcal{L}3 \to \mathcal{L}2$, and specified by: $v2 \cdot (adj \cdot s) = v2 \cdot s$, for all s in $\mathcal{L}3$.

8. "Fibolucci": Derive a program, with $\mathcal{O}(^2log \cdot N)$ time complexity, for the computation of $(\Sigma i : 0 \leq i \leq N : fib \cdot i * fib \cdot (N - i))$.

2002.6.10 2013.4.15 correction of a few simple errors

Chapter 8

Program Transformations

8.0 Transformational Programming

example: Consider functions f and g, defined by:

$$\begin{array}{lll} & f\cdot 0 & = & 3 \\ \& & f\cdot(n{+}1) & = & f\cdot n + 2 \\ \& & g\cdot n & = & 2*n + 3 \end{array}$$

On the natural numbers functions f and g are equivalent, that is, for all *natural* n we have $f\cdot n = g\cdot n$; on the negative integers, however, they are definitely not equivalent: $g\cdot n$ also is an integer for negative n, whereas $f\cdot n$ is not. This is harmless, though, if the naturals are the only domain we are interested in.

□

8.0.0 no edible recipes

8.1 Fusion

Whenever a function H is the composition of two other functions G and F, we can obtain a definition for H in two ways: either we define H explicitly as the composition of G and F, thus:

$$H \;=\; G \circ F \quad ,$$

which is fine if we have admissible definitions for G and F, or we try to *fuse* the definitions for G and F into a single, integrated definition for H. Defining H explicitly as a composition is often clearer and more modular, but fused definitions sometimes are more efficient, for example in those cases where the storage requirements of the intermediate values are large or when the construction – in F's definition – and the subsequent destruction – in G's definition – of the intermediate values is time consuming.

example: Conceptually, a compiler is composed from two components, a parser that reconstructs a parse tree from the input text and a code generator that uses the parse tree to construct the output text. By fusing these components into a single compiler we prevent that the parse tree must be constructed (and stored) in its full entirety: parts of the tree for which code already has been generated can be discarded. (Not surprisingly, this is also known as on the on-the-fly code generation.)

□

We conclude this section with a few simple examples.

8.1.0 map fusion

For functions g and f (and recalling that $\bullet$ is the operator mapping functions over lists), we can prove the following little lemma. This is an example of fusion. (The proof of this lemma is left to the exercises.)

lemma:

$$(g\bullet)\circ(f\bullet) \;=\; ((g\circ f)\bullet) \quad .$$

□

When applied from left to right this lemma reduces two list traversals to a single one.

8.1.1 the cons-snoc isomorphism, or the rev-trick

We consider a function F, of type $\mathcal{L}_*(B)\to V$, for some element type B and type V, and defined recursively, on snoc-lists, as follows:

$$\begin{array}{lll} & F\cdot[\,] & = & v0 \\ \& & F\cdot(s\triangleleft b) & = & b\oplus F\cdot s \end{array}$$

Here, $v0$ is a constant of type V and $\oplus$ is a binary operator of type $B\times V\to V$.

Suppose, now, that we wish to eliminate operator $\triangleleft$ and that we wish to redefine F as a function on cons-lists instead. This can be achieved by means of a simple transformation, which we will refer to as the "rev-trick". The trick is that we try to derive a recursive definition for the composition $F \circ rev$ of functions F and rev. Calling this composition G, we obtain as its specification:

$$G \;=\; F \circ rev \quad ,$$

which, of course, can also be formulated point-wise as:

$$G{\cdot}t \;=\; F{\cdot}(rev{\cdot}t) \quad \text{, for all lists } t \quad .$$

Using this specification and F's definition we now derive, by straightforward induction on t:

$$\begin{array}{ll}
 & G{\cdot}[\,] \\
= & \quad \{ \text{ specification of } G \text{ } \} \\
 & F{\cdot}(rev{\cdot}[\,]) \\
= & \quad \{ \text{ definition of } F \text{ } \} \\
 & v0 \quad ,
\end{array}$$

and:

$$\begin{array}{ll}
 & G{\cdot}(b \triangleright t) \\
= & \quad \{ \text{ specification of } G \text{ } \} \\
 & F{\cdot}(rev{\cdot}(b \triangleright t)) \\
= & \quad \{ \text{ definition of } rev \text{ } \} \\
 & F{\cdot}(rev{\cdot}t \triangleleft b) \\
= & \quad \{ \text{ definition of } F \text{ } \} \\
 & b \oplus F{\cdot}(rev{\cdot}t) \\
= & \quad \{ \text{ specification of } G\text{, by Induction Hypothesis } \} \\
 & b \oplus G{\cdot}t \quad .
\end{array}$$

Thus we obtain as recursive definition for G:

$$\begin{array}{lll}
 & G{\cdot}[\,] & = \; v0 \\
\& & G{\cdot}(b \triangleright t) & = \; b \oplus G{\cdot}t
\end{array}$$

Thus this definition is the result of fusing the definitions of F and rev, so it implements $F \circ rev$ (as expressed in G's specification).

The rev-trick enables us to switch easily from functions defined on snoc-lists to functions defined on cons-lists, and, of course, vice versa as well.

8.2 Tupling Revisited

In Chapter 3 we have introduced function tupling as an elementary programming technique. As a matter of fact, tupling is a form of program transformation: several functions, whose definitions exhibit the same pattern of recursion, are replaced by a single function, the values of which are tuples of several elements.

Tupling often improves the efficiency of the program: by combining several functions into one, their evaluations are *synchronised*, as a result of which needless repetitions of computation steps are avoided.

Another way of looking at it is to view tupling as a form of generalization: a function's range is extended by the addition of new elements, just as the introduction of additional parameters extends a function's domain. In this way, tupling and adding extra parameters can be regarded as complementary techniques: the one extends the function's domain whereas the other extends the function's range.

8.3 Tail Recursion and Iteration

8.3.0 linear recursion

A recursive definition of a function is called *linearly recursive* if the function's defining expression contains at most one recursive application of that function, per case distinguished in the definition. Here is the prototype of a linearly recursive definition, in which only two cases are distinguished – a *base* case and a *recursive* case –; in it function F is defined in terms of given functions b, f, g, and h, and a binary operator $\oplus$:

$$
\begin{array}{lcl}
F\cdot x & = & \mathsf{if}\ \neg b\cdot x \ \rightarrow\ f\cdot x \\
 & & [\,]\ \ \ b\cdot x \ \rightarrow\ h\cdot x \oplus F\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

Function F has type $X \rightarrow V$, for given types X and V, if b, f, g, and h have the following types:

$$
\begin{array}{ll}
b: & X \to \mathsf{Bool} \\
f: & X \to V \\
g: & X \to X \\
h: & X \to Y \\
\oplus: & Y \times V \to V
\end{array}
$$

Strictly speaking, function h is superfluous, as its effect can be included in binary operator $\oplus$. Yet, its presence allows for greater flexibility (in the choice of $\oplus$), which influences the efficiency of later programs. We will see examples of this later.

8.3.1 tail recursion

A linearly recursive definition is called *tail recursive* if, per case distinguished in the definition, the function's recursive application is the main operator in the function's defining expression; that is, the recursive application is the root of the tree represented by that expression. Here is the prototype of a tail-recursive definition of a function G, in terms of given functions b, f, and g:

$$
\begin{array}{lll}
G\cdot x & = & \mathsf{if}\ \neg b\cdot x \to f\cdot x \\
 & & [\!]\ \ b\cdot x \to G\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

A tail-recursive definition in which two recursive cases are distinguished has this general shape:

$$
\begin{array}{lll}
G2\cdot x & = & \mathsf{if}\ \neg(b0\cdot x \vee b1\cdot x) \to f\cdot x \\
 & & [\!]\ \ b0\cdot x \to G2\cdot(g0\cdot x) \\
 & & [\!]\ \ b1\cdot x \to G2\cdot(g1\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

8.3.2 iteration

Tail recursive definitions are of interest because they correspond to iteration in sequential programming in a very straightforward way. An application of a function with a tail-recursive definition can be simply implemented as a repetition in a sequential program. With function G as defined above, the following program fragment establishes $v = G\cdot x0$, for some input value $x0$ and output variable v; in addition, a local variable x is used:

sequential program i :

$$
\begin{array}{ll}
 & x := x0 \\
; & \{ \text{ invariant: } G\cdot x0 = G\cdot x \ \} \\
 & \text{do } b\cdot x \ \rightarrow \ x := g\cdot x \text{ od} \\
; & \{ \ G\cdot x0 = f\cdot x \ \} \\
 & v := f\cdot x \\
 & \{ \ v = G\cdot x0 \ \}
\end{array}
$$

□

Termination of the repetition is guaranteed if (and only if) G's tail recursive definition is well-founded, which, in this case, boils down to the requirement that the infinite sequence $x0$, $g\cdot x0$, $g\cdot(g\cdot x0)$, $\cdots$ contains at least one element for which boolean function b has value false.

8.4 Tail fusion

Fusion of function definitions is particularly simple if the definition of the right-hand side function – in the composition – is tail-recursive.

We recall the prototype of a tail-recursive definition of a function G, in terms of given functions b, f, and g:

$$
\begin{array}{lll}
G\cdot x & = & \text{if } \neg b\cdot x \ \rightarrow \ f\cdot x \\
 & & [] \quad b\cdot x \ \rightarrow \ G\cdot(g\cdot x) \\
 & & \text{fi}
\end{array}
$$

Suppose that, for some given function F, we are interested in the composition $F \circ G$. We call this composition H and its (point-wise) specification is:

$$H\cdot x \ = \ F\cdot(G\cdot x) \quad \text{, for all } x \quad .$$

Now we derive, by straightforward Mathematical Induction on G's domain:

$$
\begin{array}{ll}
 & H\cdot x \\
= & \quad \{ \text{ specification of } H \ \} \\
 & F\cdot(G\cdot x) \\
= & \quad \{ \text{ definition of } G \text{, case } \neg b\cdot x \ \} \\
 & F\cdot(f\cdot x)
\end{array}
$$

and:

$$
\begin{array}{ll}
 & H\cdot x \\
= & \quad \{ \text{ specification of } H \ \} \\
 & F\cdot(G\cdot x) \\
= & \quad \{ \text{ definition of } G \text{, case } b\cdot x \ \} \\
 & F\cdot(G\cdot(g\cdot x)) \\
= & \quad \{ \text{ specification of } H \text{ , by Induction Hypothesis } \} \\
 & H\cdot(g\cdot x) \quad .
\end{array}
$$

From this derivation we obtain an equally tail-recursive definition for H:

$$
\begin{array}{lll}
H\cdot x & = & \mathsf{if} \ \neg b\cdot x \ \rightarrow \ F\cdot(f\cdot x) \\
 & & [] \quad b\cdot x \ \rightarrow \ H\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

Notice that this definition differs from G's definition *only* in the base case, where $f\cdot x$ is replaced by $F\cdot(f\cdot x)$. Hence,the composition of F with a tail-recursively defined function G boils down the composition of F with function f in the base case of G's definition: f is just replaced by $F\circ f$.

* * *

Similarly, the application $f\cdot x$ equals $f\cdot(I\cdot x)$, so f is the composition of f with the identity function I. Consequently, the above definition of function G itself can be considered as the result of fusing f with the tail-recursive definition of a (slightly simpler) function $G0$, say, defined by:

$$
\begin{array}{lll}
G0\cdot x & = & \mathsf{if} \ \neg b\cdot x \ \rightarrow \ x \\
 & & [] \quad b\cdot x \ \rightarrow \ G0\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

With this $G0$ we then have $G = f\circ G0$.

Tail-fusion does not necessarily yield more efficient programs, but recognizing that a definition is a fusion result may enhance clarity, as this makes explicit that the function is the composition of simpler functions, for example, as in $G = f\circ G0$. Thus, "unfusion" contributes to our understanding of the function's structure.

Finally, because function composition is associative, tail-fusion can be repeated as often as desired; for instance, the base case $G\cdot(f\cdot x)$ in H's definition equals $(G\circ f)\cdot x$, which matches the pattern of the prototype again.

8.5 From Linear to Tail Recursion I

We consider a linearly recursive definition of the following shape; as before, function F is defined in terms of given functions b, f, g, and h, and a binary operator $\oplus$. This is a special case of the prototyype in Section 8.3.0, in that type Y now equals V; as a result, operator $\oplus$ now has type $V \times V \to V$:

$$
\begin{array}{lcl}
F \cdot x & = & \mathsf{if}\ \neg b \cdot x \ \to\ f \cdot x \\
 & & [\!]\ \ b \cdot x \ \to\ h \cdot x \oplus F \cdot (g \cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

Function F has type $X \to V$, as we assume b, f, g, and h to have the following types:

$$
\begin{array}{l}
b : X \to \mathsf{Bool} \\
f : X \to V \\
g : X \to X \\
h : X \to V \\
\oplus : V \times V \to V
\end{array}
$$

If binary operator $\oplus$ has a left identity element e, say:

(0) $\quad e \oplus v \ = \ v$, for all v in V ,

then we can rewrite $F \cdot x$ as $e \oplus F \cdot x$ and, hence, we can view both $F \cdot x$ and $h \cdot x \oplus F \cdot (g \cdot x)$ as instances of a more general expression, of the shape $u \oplus F \cdot x$. By standard generalization, we introduce a new function G, of type $V \to X \to V$, and with this specification:

(1) $\quad G \cdot u \cdot x \ = \ u \oplus F \cdot x$, for all u and x .

In terms of G (and $\oplus$'s identity e) function F can now be defined by:

$$F \cdot x \ = \ G \cdot e \cdot x \ ,$$

and this is meaningful provided we can derive a definition for G in which F does not occur. (Notice that G's specification, (1), can be considered a definition as well, but one that contains F.) To obtain such a definition we follow the case analysis in F's definition and we use mathematical induction on x; so, we derive, for the case $\neg b \cdot x$:

$$
\begin{array}{ll}
 & G\cdot u\cdot x \\
= & \quad \{ \text{ specification } (1)\text{ , of } G \text{ } \} \\
 & u \oplus F\cdot x \\
= & \quad \{ \text{ definition of } F\text{ , case } \neg b\cdot x \text{ } \} \\
 & u \oplus f\cdot x \quad ,
\end{array}
$$

and, for the other case, $b\cdot x$:

$$
\begin{array}{ll}
 & G\cdot u\cdot x \\
= & \quad \{ \text{ specification } (1)\text{ , of } G \text{ } \} \\
 & u \oplus F\cdot x \\
= & \quad \{ \text{ definition of } F\text{ , case } b\cdot x \text{ } \} \\
 & u \oplus (h\cdot x \oplus F\cdot(g\cdot x)) \\
= & \quad \{ \bullet \text{ assume that } \oplus \text{ is associative, see (2) below } \} \\
 & (u \oplus h\cdot x) \oplus F\cdot(g\cdot x) \\
= & \quad \{ \text{ specification } (1)\text{ , of } G\text{, by Induction Hypothesis } \} \\
 & G\cdot(u \oplus h\cdot x)\cdot(g\cdot x) \quad .
\end{array}
$$

This derivation is based on an additional algebraic property of $\oplus$, namely that it is *associative*, which means:

(2) $\quad u \oplus (v \oplus w) \;=\; (u \oplus v) \oplus w$, for all u, v, w in V .

With these properties we obtain the following recursive definition for G, which happens to be tail recursive:

$$
\begin{array}{lll}
G\cdot u\cdot x & = & \mathsf{if}\ \ \neg b\cdot x \ \rightarrow\ u \oplus f\cdot x \\
 & & [\,]\quad\ \ b\cdot x \ \rightarrow\ G\cdot(u \oplus h\cdot x)\cdot(g\cdot x) \\
 & & \mathsf{fi}
\end{array}
$$

As a result, we have obtained the

first tail-recursion theorem: for binary operator $\oplus$ that is associative and has a left identity element e.

if: $$F\cdot x \;=\; \begin{array}{ll} \mathsf{if} & \neg b\cdot x \;\rightarrow\; f\cdot x \\ [] & b\cdot x \;\rightarrow\; h\cdot x \oplus F\cdot(g\cdot x) \\ \mathsf{fi} & \end{array}$$

then: $$G\cdot u\cdot x \;=\; u \oplus F\cdot x$$

with as special case:

$$F\cdot x \;=\; G\cdot e\cdot x$$

where: $$G\cdot u\cdot x \;=\; \begin{array}{ll} \mathsf{if} & \neg b\cdot x \;\rightarrow\; u \oplus f\cdot x \\ [] & b\cdot x \;\rightarrow\; G\cdot(u\oplus h\cdot x)\cdot(g\cdot x) \\ \mathsf{fi} & \end{array}$$

This theorem has a *mirror image*: for defining expressions for the recursive case in F's definition of the shape $F\cdot(g\cdot x) \oplus h\cdot x$, instead of $h\cdot x \oplus F\cdot(g\cdot x)$, we can formulate a similar theorem for binary operators $\oplus$ with a *right* identity element, simply by mirroring the expressions in specification and definition for G as well.

The first tail-recursion theorem is not very deep, but it encapsulates a common pattern of reasoning. Preferably, the theorem is not to be learnt by heart but the underlying pattern of reasoning is to be well understood: then a similar pattern of reasoning can be applied in situations where the theorem is not directly applicable, because of minor deviations. In such cases it often is less labour to redo the generalization and derivations than to forge the problem at hand into the mould of the theorem.

* * *

In many applications of the tail-recursion theorem function value $f\cdot x$ is a constant, namely the right identity element of operator $\oplus$. This gives rise to the following special case of the theorem. Because now $f\cdot x$ equals e, the expression $u\oplus f\cdot x$ in G's definition collapses to just u:

special case of the first tail-recursion theorem: for binary operator $\oplus$ that is associative and has a (left and right) identity element e.

if:
$$F\cdot x \;=\; \mathsf{if}\;\; \neg b\cdot x \;\rightarrow\; e \;\;[\!]\;\; b\cdot x \;\rightarrow\; h\cdot x \oplus F\cdot(g\cdot x) \;\;\mathsf{fi}$$

then:
$$G\cdot u\cdot x \;=\; u \oplus F\cdot x$$

with as special case:
$$F\cdot x \;=\; G\cdot e\cdot x$$

where:
$$G\cdot u\cdot x \;=\; \mathsf{if}\;\; \neg b\cdot x \;\rightarrow\; u \;\;[\!]\;\; b\cdot x \;\rightarrow\; G\cdot(u\oplus h\cdot x)\cdot(g\cdot x) \;\;\mathsf{fi}$$

In this form the theorem is most widely "known" in the functional programming folklore. In this folklore, parameter u is often called an *accumulating parameter*, because it eventually represents G's value – via its base case – and because in this parameter G's value is gradually "accumulated" – via the expression $u\oplus h\cdot x$ in its recursive case –.

This is all very well, but it is important to remember that we have derived the theorem – both the general one and the special case – by means of the standard generalization technique, without using any operational connotations. So, the moral of this story is that transformations like these can be performed by just "letting the symbols do the work".

* * *

A sequential-program fragment for the computation of $F\cdot x0$ is obtained by application of the standard implementation scheme for tail recursive definitions. Because now function G has two parameters two local variables are used in this program, one for each parameter. The repetition invariant is

$$G\cdot e\cdot x0 \;=\; g\cdot u\cdot x \quad .$$

Via G's specification, (1), this invariant can also be formulated in terms of our original function F as:

$$F\cdot x0 \;=\; u\oplus F\cdot x \quad ,$$

which, not surprisingly, in sequential programming is known as a *tail invariant*.

sequential program ii :

$$
\begin{array}{ll}
 & u, x := e, x0 \\
; & \{ \text{ invariant}: \ F{\cdot}x0 \ = \ u \oplus F{\cdot}x \ \} \\
 & \text{do} \ \ b{\cdot}x \ \ \rightarrow \ \ u, x \ := \ u \oplus h{\cdot}x \, , \, g{\cdot}x \ \ \text{od} \\
; & \{ \ F{\cdot}x0 \ = \ u \oplus f{\cdot}x \ \} \\
 & v := u \oplus f{\cdot}x \\
 & \{ \ v \ = \ F{\cdot}x0 \ \}
\end{array}
$$

□

* * *

We conclude this section with a discussion of the role of the seemingly superfluous function h in the prototype of a linearly recursive definition, with which we began this section:

$$
\begin{array}{lll}
F{\cdot}x & = & \text{if} \ \ \neg b{\cdot}x \ \rightarrow \ \ f{\cdot}x \\
 & & [] \ \ \ \ b{\cdot}x \ \rightarrow \ \ h{\cdot}x \oplus F{\cdot}(g{\cdot}x) \\
 & & \text{fi}
\end{array}
$$

After all, the effect of function h can be accounted for by a proper choice of binary operator $\oplus$; that is, if so desired, we can define a new operator $\oplus'$, say, for all x, v:

$$
x \oplus' v \ = \ h{\cdot}x \oplus v \ \ .
$$

Now we can be reformulate F's definition as follows:

$$
\begin{array}{lll}
F{\cdot}x & = & \text{if} \ \ \neg b{\cdot}x \ \rightarrow \ \ f{\cdot}x \\
 & & [] \ \ \ \ b{\cdot}x \ \rightarrow \ \ x \oplus' F{\cdot}(g{\cdot}x) \\
 & & \text{fi}
\end{array}
$$

Thus, we have eleminated h from the prototype and, hence, it is redundant. So, what is its purpose?

The answer is: to enlarge our manipulative freedom. The first tail-recursion theorem requires $\oplus$ to have certain algebraic properties. The redundancy provided for by the presence of function h will make it easier to choose $\oplus$ in such a way that it has the desired algebraic properties indeed.

8.5.0 computational consequences of the theorem

We reconsider functions F and G, as defined above and we investigate how $F \cdot x0$ is evaluated, for a given value $x0$ of type X. For this purpose we introduce an infinite sequence s, defined as follows, for all natural i:

$$s_0 = x0 \ \wedge \ s_{i+1} = g \cdot s_i \quad .$$

So, $s_i = g^i \cdot x0$. In addition, let n be the *smallest natural* satisfying $\neg b \cdot s_n$; formally, this means that n is the (unique) natural satisfying:

$$(\forall i : 0 \leq i < n : b \cdot s_i) \ \wedge \ \neg b \cdot s_n \quad .$$

The computation, by normal order reduction of $F \cdot x0$, now proceeds in the following way – with s_0 substituted for $x0$ and assuming n to be "large enough"–:

$$\begin{array}{ll}
 & F \cdot s_0 \\
= & \quad \{ \text{ definition of } F \text{, assuming } b \cdot s_0 \text{ (because } n \text{ is large enough) } \} \\
 & h \cdot s_0 \oplus F \cdot s_1 \\
= & \quad \{ \text{ definition of } F \text{, assuming } b \cdot s_1 \text{ once more } \} \\
 & h \cdot s_0 \oplus (h \cdot s_1 \oplus F \cdot s_2) \\
= & \quad \{ \text{ and so on } \ldots \ \} \\
 & h \cdot s_0 \oplus (h \cdot s_1 \oplus \ \cdots (h \cdot s_{n-1} \oplus F \cdot s_n) \cdots) \\
= & \quad \{ \text{ definition of } F \text{, using } \neg b \cdot s_n \ \} \\
 & h \cdot s_0 \oplus (h \cdot s_1 \oplus \ \cdots (h \cdot s_{n-1} \oplus f \cdot s_n) \cdots) \quad .
\end{array}$$

The expression thus obtained contains n occurrences of operator $\oplus$, which can only be evaluated "inside out": $h \cdot s_0 \oplus (\cdots)$ (generally) cannot be evaluated until the right-hand side $(\cdots)$ has been evaluated first. The only subexpression that is reducible to start with is the one with the innermost occurrence of $\oplus$, that is, $h \cdot s_{n-1} \oplus f \cdot s_n$. So, in this expression the occurrences of $\oplus$ can be evaluated "from right to left" only, so to speak.

Notice that evaluation of this expression can only begin after it has been fully constructed, because the redex $h \cdot s_{n-1} \oplus f \cdot s_n$ only emerges in the last step of the above calculation. As a consequence, in addition to its time complexity, the space complexity of the computation of $F \cdot s_0$ also is at least proportional to n.

* * *

Now we consider the computation of the same value $F \cdot s_0$, but this time by means of function G, according tot the first tail-recursion theorem:

$$
\begin{array}{ll}
 & F \cdot s_0 \\
= & \quad \{ \text{ definition of } F \text{, in terms of } G \ \} \\
 & G \cdot e \cdot s_0 \\
= & \quad \{ \text{ definition of } G \text{, (again) assuming } b \cdot s_0 \ \} \\
 & G \cdot (e \oplus h \cdot s_0) \cdot s_1 \\
= & \quad \{ \text{ definition of } G \text{, assuming } b \cdot s_1 \text{ once more } \} \\
 & G \cdot ((e \oplus h \cdot s_0) \oplus h \cdot s_1) \cdot s_2 \\
= & \quad \{ \text{ and so on } \ldots \ \} \\
 & G \cdot ((\cdots((e \oplus h \cdot s_0) \oplus h \cdot s_1) \cdots) \oplus h \cdot s_{n-1}) \cdot s_n \\
= & \quad \{ \text{ definition of } G \text{, using } \neg b \cdot s_n \ \} \\
 & ((\cdots((e \oplus h \cdot s_0) \oplus h \cdot s_1) \cdots) \oplus h \cdot s_{n-1}) \ \oplus \ f \cdot s_n \quad .
\end{array}
$$

The expression thus obtained contains $n+1$ occurrences of operator $\oplus$, but this time the only applicable evaluation order is "from left to right": the only redex to start with is $e \oplus h \cdot s_0$. So, we conclude that application of the first tail-recursion theorem *reverses* the order in which the occurrences of $\oplus$ will be evaluated.

Moreover, the redex $e \oplus h \cdot s_0$ already appears in the very first step (in which G's definition is applied). Normal-order reduction, however, will still construct the whole expression (with all $n+1$ occurrences of $\oplus$), so normal-order reduction still requires $\mathcal{O}(n)$ space to store this whole expression. By means of a different reduction strategy, however, the use of storage space can now be reduced to $\mathcal{O}(1)$, namely by evaluating every $\oplus$ as soon as it emerges, which, in this case, is possible. The evaluation of $F \cdot s_0$ then would proceed as follows:

$$
\begin{array}{ll}
 & F \cdot s_0 \\
= & \quad \{ \text{ definition of } F \text{, in terms of } G \ \} \\
 & G \cdot e \cdot s_0 \\
= & \quad \{ \text{ definition of } G \text{, assuming } b \cdot s_0 \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & G\cdot(e\oplus h\cdot s_0)\cdot s_1 \\
= & \quad \{ \text{ evaluation of } \oplus \text{, call the result } u_0 \ \} \\
 & G\cdot u_0\cdot s_1 \\
= & \quad \{ \text{ definition of } G \text{, assuming } b\cdot s_1 \ \} \\
 & G\cdot(u_0\oplus h\cdot s_1)\cdot s_2 \\
= & \quad \{ \text{ evaluation of } \oplus \text{, call the result } u_1 \ \} \\
 & G\cdot u_1\cdot s_2 \\
= & \quad \{ \text{ and so on } \ldots \ \} \\
 & G\cdot u_{n-1}\cdot s_n \\
= & \quad \{ \text{ definition of } G \text{, using } \neg b\cdot s_n \ \} \\
 & u_{n-1} \oplus f\cdot s_n \\
= & \quad \{ \text{ evaluation of } \oplus \text{, call the result } u_n \ \} \\
 & u_n \quad .
\end{array}
$$

During this computation the sizes of the intermediate expressions do not increase and, hence, remain $\mathcal{O}(1)$. Thus, the first tail-recursion theorem can also be used to reduce the storage complexity of a function.

remark: To what extent this saving of storage space is really obtainable depends very much on how the functional language is implemented. Some compilers perform a, so-called, *strictness analysis* in order to determine whether the required deviation from standard normal-order reduction is safe: too naive such deviations may give rise to non-terminating computations! Other compilers may require the programmer to supply hints about the evaluation strategy to be used. In the above example, for instance, a hint may given to indicate that G's first parameter, u, may safely be evaluated *eagerly*, that is, as soon as possible.

□

Notice that if the first tail-recursion theorem is used to construct a sequential program, the saving of storage space is obtained for free: sequential program ii – at the end of the previous subsection – uses $\mathcal{O}(1)$ storage space only.

8.5.1 example: summing a function

Let h be a given function, of type $\mathsf{Nat} \to \mathsf{Int}$. For given natural constant N we are interested in the sum of the "first" N values of h, that is, we are interested in $(\Sigma i : 0 \le i < N : h \cdot i)$. Replacing constant N by a variable n we introduce a function F, with specification:

$$F \cdot n \;=\; (\Sigma i : 0 \le i < n : h \cdot i) \quad ,$$

as a result of which $(\Sigma i : 0 \le i < N : h \cdot i) = F \cdot N$, and for which we can derive this linearly recursive definition:

$$\begin{array}{lll} & F \cdot 0 & = & 0 \\ \& & F \cdot (n{+}1) & = & F \cdot n + h \cdot n \end{array}$$

Because addition has 0 as its (left and right) identity element and because it is associative, we can apply the (mirrored) first tail-recursion theorem. That is, we introduce a function G, with specification:

$$G \cdot x \cdot n \;=\; h \cdot n + x \quad ,$$

and the first tail-recursion then tells us that:

$$(\Sigma i : 0 \le i < N : h \cdot i) \;=\; G \cdot 0 \cdot N \quad ,$$

provided we define G by:

$$\begin{array}{lll} & G \cdot x \cdot 0 & = & x \\ \& & G \cdot x \cdot (n{+}1) & = & G \cdot (h \cdot n + x) \cdot n \end{array}$$

Thus, by applying the scheme from sequential program ii, we obtain a sequential program for the computation of $(\Sigma i : 0 \le i < N : h \cdot i)$.

sequential program for summation:

$$\begin{array}{ll} & x, n := 0, N \\ ; & \{\ \mathsf{invariant}:\ F \cdot N = x + F \cdot n\ \} \\ & \mathsf{do}\ \ n \ne 0\ \ \to\ \ x, n := h \cdot (n{-}1) + x\,,\ n{-}1\ \ \mathsf{od} \\ & \{\ x = F \cdot N\ \} \end{array}$$

□

* * *

To illustrate once more the important role of function h in obtaining the required associativity of the binary operator, we consider the following special case: $h \cdot n = n^2$. The definition of function F then becomes:

$$\begin{array}{lll} & F \cdot 0 & = & 0 \\ \& & F \cdot (n+1) & = & F \cdot n + n^2 \end{array}$$

We could, however, also introduce a binary operator $\oplus$, defined by, for all integer x, y:

$$x \oplus y \;=\; x + y^2 \quad ,$$

and then we could rewrite F's definition:

$$\begin{array}{lll} & F \cdot 0 & = & 0 \\ \& & F \cdot (n+1) & = & F \cdot n \oplus n \end{array}$$

This operator $\oplus$ is, however, not associative anymore and the first tail-recursion theorem cannot be applied directly here.

8.5.2 example: list reversal

A simple recursive definition for function rev, mapping finite lists to their reversed versions, is – obtained from Chapter 6 –:

$$\begin{array}{lll} & rev \cdot [\,] & = & [\,] \\ \& & rev \cdot (b \triangleright t) & = & rev \cdot t \mathbin{+\!\!+} [b] \end{array}$$

Binary operator $+\!\!+$ has $[\,]$ as its right-identity element and is associative, so we can generalize rev to a function G with this specification, for all lists s, t:

$$G \cdot s \cdot t \;=\; rev \cdot t \mathbin{+\!\!+} s \quad .$$

In terms of G function rev can now be defined by:

$$rev \cdot t \;=\; G \cdot [\,] \cdot t \quad ,$$

and a tail-recursive definition for G is easily obtained:

$$\begin{array}{lll} & G \cdot s \cdot [\,] & = & [\,] \mathbin{+\!\!+} s \\ \& & G \cdot s \cdot (b \triangleright t) & = & G \cdot ([b] \mathbin{+\!\!+} s) \cdot t \end{array}$$

By using some elementary properties of the list operators we can simplify this definition a little:

$$\begin{array}{lll} & G\cdot s\cdot[\,] & = & s \\ \& & G\cdot s\cdot(b\triangleright t) & = & G\cdot(b\triangleright s)\cdot t \end{array}$$

This transformation has a dramatic effect on the program's efficiency: because the time complexity of $+\!\!+$ is *linear* in the length of its left argument, the above naive definition for *rev* is quadratic, whereas (both versions of) G's definition has linear time complexity. This is because now $+\!\!+$ only has $[b]$ as its left argument, with length 1. (And, therefore, $[b]+\!\!+s$ can be recoded as $b\triangleright s$.)

8.6 More Complicated Algebraic Patterns

If the defining expression of a function displays a more complicated algebraic structure than just an associative operator, it may still be possible to transform such a definition into tail-recursive form. By just giving it a try we seek to discover what algebraic properties the operators in the expression should have in order for the transformation to be successful. Again, generalization is the technique to be used. Here is an example.

From Section 3.9.1 we recall a function g with this specification:

(3) $\quad g\cdot x\cdot n \;=\; (\Sigma i : 0\le i<n : x^i)$, for natural n .

In that very section we have derived this recursive definition for g:

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

Addition is certainly associative and it has 0 for its identity element, so both defining expressions for the two recursive cases, as well as $g\cdot x\cdot n$ itself, can be viewed as expressions of the shape $z+$ "something containing $g\cdot x\cdot n$" . This "something containing $g\cdot x\cdot n$" is not just $g\cdot x\cdot n$ itself, because of the multiplications with x and $(1+x)$ in the two recursive cases, respectively. To account for these multiplications we introduce yet another parameter, y; thus, we observe that the general shape of the expressions is $z + y * g\cdot x\cdot n$. This is also true of $g\cdot x\cdot n$ itself, because multiplication has 1 as its (left) identity element, so we have:

$$g\cdot x\cdot n \;=\; 0 + 1 * g\cdot x\cdot n \quad .$$

Thus, we generalize function g to a new function G, with this specification:

(4) $\quad G{\cdot}z{\cdot}y{\cdot}x{\cdot}n \;=\; z + y * g{\cdot}x{\cdot}n$, for all z, y, x, and n .

So, by construction, in terms of G function g can now be defined thus:

$$g{\cdot}x{\cdot}n \;=\; G{\cdot}0{\cdot}1{\cdot}x{\cdot}n \;\;,$$

where the algebraic properties needed to justify this are that both addition and multiplication have (left) identities, 0 and 1 respectively.

Using specification (4) (as the proof obligation) and the above definition of g (as a fact), we derive a recursive definition for G as follows, again by case analysis and induction on n:

$$\begin{array}{ll}
 & G{\cdot}z{\cdot}y{\cdot}x{\cdot}0 \\
= & \quad \{ \text{ specification, (4), of } G \text{ } \} \\
 & z + y * g{\cdot}x{\cdot}0 \\
= & \quad \{ \text{ definition of } g \text{ } \} \\
 & z + y * 0 \\
= & \quad \{ \text{ } \bullet \text{ algebra, see below } \} \\
 & z \;\;.
\end{array}$$

In the step labelled "$\bullet$ algebra" we have not only used that 0 is the identity element of addition but also that it is a *zero element* of multiplication:

$$y * 0 \;=\; 0 \;\;, \text{ for all } y \;\;.$$

We continue our derivation, for the next case:

$$\begin{array}{ll}
 & G{\cdot}z{\cdot}y{\cdot}x{\cdot}(2{*}n{+}1) \\
= & \quad \{ \text{ specification, (4), of } G \text{ } \} \\
 & z + y * g{\cdot}x{\cdot}(2{*}n{+}1) \\
= & \quad \{ \text{ definition of } g \text{ } \} \\
 & z + y * (1 + x * g{\cdot}x{\cdot}(2{*}n)) \\
= & \quad \{ \text{ } \bullet \text{ algebra, see below } \} \\
 & (z{+}y) + (y{*}x) * g{\cdot}x{\cdot}(2{*}n) \\
= & \quad \{ \text{ specification, (4), of } G\text{, by Induction Hypothesis } \} \\
 & G{\cdot}(z{+}y){\cdot}(y{*}x){\cdot}x{\cdot}(2{*}n) \;\;.
\end{array}$$

In the step labelled "• algebra" several algebraic properties of $+$ and $*$ play a role: in addition to both being associative, we have also used that 1 is the right identity of $*$ and that $*$ *distributes over* $+$, that is:

$$y*(z+x) \;=\; y*z + y*x \quad \text{, for all } x,y,z \quad .$$

We continue our derivation, for the last case:

$$\begin{array}{ll}
 & G\cdot z\cdot y\cdot x\cdot(2*n+2) \\
= & \quad \{ \text{ specification, } (4)\text{, of } G \text{ } \} \\
 & z + y*g\cdot x\cdot(2*n+2) \\
= & \quad \{ \text{ definition of } g \text{ } \} \\
 & z + y*(\,(1+x)*g\cdot(x*x)\cdot(n+1)\,) \\
= & \quad \{ \text{ algebra } \} \\
 & z + (\,y*(1+x)\,)*g\cdot(x*x)\cdot(n+1) \\
= & \quad \{ \text{ specification, } (4)\text{, of } G\text{, by Induction Hypothesis } \} \\
 & G\cdot z\cdot(y*(1+x))\cdot(x*x)\cdot(n+1) \quad .
\end{array}$$

In this derivation the only algebraic property needed is $*$'s associativity.

By collecting the results of these derivations we obtain as definition for G:

$$\begin{array}{lll}
 & G\cdot z\cdot y\cdot x\cdot 0 & = & z \\
\& & G\cdot z\cdot y\cdot x\cdot(2*n+1) & = & G\cdot(z+y)\cdot(y*x)\cdot x\cdot(2*n) \\
\& & G\cdot z\cdot y\cdot x\cdot(2*n+2) & = & G\cdot z\cdot(y*(1+x))\cdot(x*x)\cdot(n+1)
\end{array}$$

Aside: Algebraists call the structure underlying the first tail-recursion theorem a *semi-group*, whereas the algebraic structure needed for the example in this subsection is a *ring with identity*. This also shows that the solution for this example is applicable to any such structure, as we have used no other algebraic properties of $+$ and $*$. The type of parameter x, for instance, may be "square matrix" instead of just "number"; then, $+$ and $*$ denote matrix addition and multiplication respectively.

□

The above tail-recursive definition for G can be used for the construction of a sequential program for the computation of a value $g\cdot X\cdot N$ (which equals $(\,\Sigma i : 0\le i<X : N^i\,)\,)$. To this end, we first have to eliminate the parameter patterns, according to the rules of the game:

$$
\begin{array}{lllll}
G\cdot z\cdot y\cdot x\cdot m & = & \mathsf{if} & m=0 & \rightarrow\ z \\
 & & [] & m\neq 0 \wedge m\,\mathsf{mod}\,2\neq 0 & \rightarrow\ G\cdot(z+y)\cdot(y*x)\cdot x\cdot(m-1) \\
 & & [] & m\neq 0 \wedge m\,\mathsf{mod}\,2=0 & \rightarrow\ G\cdot z\cdot(y*(1+x))\cdot(x*x)\cdot(m\,\mathsf{div}\,2) \\
 & & \mathsf{fi} & &
\end{array}
$$

Now the construction of a sequential program is straightforward, if we use the guarded-command language. (To avoid the use of guarded commands, one uses a simple repetition of the shape $\mathsf{while}\ m\neq 0\ \mathsf{do}\ \cdots\ \mathsf{od}$, with a selection in its body to distinguish the cases $m\,\mathsf{mod}\,2\neq 0$ and $m\,\mathsf{mod}\,2=0$):

sequential program:

$$
\begin{array}{ll}
 & \{\ 0\leq N\ \} \\
 & z,y,x,m\ :=\ 0,1,X,N \\
;\ & \{\ \mathsf{invariant}:\ g\cdot x\cdot N\ =\ z\ +\ y*g\cdot x\cdot m\ \wedge\ 0\leq m\ \} \\
 & \mathsf{do}\ \ m\neq 0 \wedge m\,\mathsf{mod}\,2\neq 0\ \rightarrow\ \ z,y,m\ :=\ z+y\,,\ y*x\,,\ m-1 \\
 & []\ \ m\neq 0 \wedge m\,\mathsf{mod}\,2=0\ \rightarrow\ \ y,x,m\ :=\ y*(1+x)\,,\ x*x\,,\ m\,\mathsf{div}\,2 \\
 & \mathsf{od} \\
 & \{\ z\ =\ g\cdot x\cdot N\ ,\ \text{hence, by (3):}\ \ z\ =\ (\Sigma i: 0\leq i<N: X^i)\ \}
\end{array}
$$

□

With this program $(\Sigma i: 0\leq i<N: X^i)$ can be computed in $\mathcal{O}(log\cdot N)$ time. As is the case with the functional program – that is, the above definition of function g – we used as a starting point, this program is totally incomprehensible when viewed in isolation. We can only understand "what this program does" by taking into account how it has been constructed.

8.7 Operator Folding

As a further illustration of the generalization technique we reinvent the concept of *folded operators*, which are also discussed extensively in R.S. Bird's textbook [1]. We will use this to obtain a general-purpose theorem for the transformation of *linear recursion* into *tail recursion*. In addition, we will apply this theorem to derive an evaluator for postfix expressions smoothly; this example illustrates how definitions with multiple recursion can be transformed into tail-recursive form.

* * *

We consider a binary operator $\oplus$ of type $B\times V\rightarrow V$, for some types B and V. In this section dummies b and c have type B whereas v has type V.

With $\oplus$ we can form expressions of type V, like:

$$v \quad , \quad b \oplus v \quad , \quad b \oplus (c \oplus v) \quad \text{, and so on .}$$

These expressions have in common that they depend on *one* value of type V and *some* –zero or more– values of type B. The order of the B values is relevant, so we can consider them as the elements of a *list* of type $\mathcal{L}_*(B)$. We now rewrite the above expressions in terms of a single $\mathcal{L}_*(B)$ and a V, by introducing a new binary operator $\otimes$, of type $\mathcal{L}_*(B) \times V \to V$, and which we require to satisfy:

$$\begin{array}{rcr} v & = & [\,] \otimes v \\ b \oplus v & = & [b] \otimes v \\ b \oplus (c \oplus v) & = & [b, c] \otimes v \end{array}$$

Moreover, because $b \oplus (c \oplus v) = (b \oplus v)(v := c \oplus v)$, our new operator must, for the sake of consistency, also satisfy:

$$[b, c] \otimes v \;=\; [b] \otimes (c \oplus v) \quad .$$

From this latter relation we obtain, by a simple generalization –replace the singleton list $[b]$ by any list s–, the following recursive definition for $\otimes$:

$$\begin{array}{rcl} [\,] \otimes v & = & v \\ (s \lhd c) \otimes v & = & s \otimes (c \oplus v) \end{array}$$

Operator $\otimes$ has other interesting properties, that can be proved by straightforward Mathematical Induction (over the size of t):

$$\begin{array}{rcl} (b \rhd t) \otimes v & = & b \oplus (t \otimes v) \\ (s +\!\!+ t) \otimes v & = & s \otimes (t \otimes v) \end{array}$$

An operator like $\otimes$ is well-known in the functional-programming community: Bird calls it $\mathit{foldr}\,(\oplus)$, but in calculational derivations a short, ad hoc, name like $\otimes$ is more manageable than a long expression like $\mathit{foldr}\,(\oplus)$. (In addition there is no point in dragging around the constant parameter $\oplus$.)

In $\otimes$'s type, $\mathcal{L}_*(B) \times V \to V$, type V occurs to the *right* of the list type; therefore $\otimes$ is called *right-folded* $\oplus$. In a similar way, for binary operators of type $V \times B \to V$, corresponding left-folded operators can be constructed, of type $V \times \mathcal{L}_*(B) \to V$.

The above shows that folded operators can be invented without any appeal to their usual operational connotations, just by isolating the common pattern from a few similar expressions.

example: Function application is a binary operator and we can apply the above to it: with $B := (V \to V)$ function application indeed has type $B \times V \to V$. So, introducing $\odot$ for right-folded $\cdot$ we obtain as definition for $\odot$:

$$\begin{array}{rcl} [\,] \odot v & = & v \\ (s \triangleleft g) \odot v & = & s \odot (g \cdot v) \quad , \end{array}$$

with properties like:

$$[\,f, g\,] \odot v \;=\; f \cdot (g \cdot v) \quad .$$

Hence, a list s of type $\mathcal{L}_*(V \to V)$ now represents the continued composition of its (function) elements and $\odot$ represents continued function application: $s \odot v$ is the application to v of all functions in s. Moreover, this example shows that it helps to have an explicit symbol for function application: Function application really is a binary operation just like any other.

□

8.8 From Linear to Tail Recursion II

We consider the following linearly recursive definition of a function F, in terms of given functions b, f, g, and h, and a binary operator $\oplus$:

$$\begin{array}{rcl} F \cdot x & = & \mathsf{if}\ \neg b \cdot x \to f \cdot x \\ & & [\!]\ \ b \cdot x \to h \cdot x \oplus F \cdot (g \cdot x) \\ & & \mathsf{fi} \end{array}$$

Function F has type $X \to V$, for some types X and V, provided b, f, g, and h have the following types:

$$\begin{array}{l} b : X \to \mathsf{Bool} \\ f : X \to V \\ g : X \to X \\ h : X \to Y \\ \oplus : Y \times V \to V \end{array}$$

As before, function h is formally superfluous, as its effect can be included in the binary operator $\oplus$. Yet, its presence allows for greater flexibility, which may influence the efficiency of later programs. We will illustrate this.

With $\otimes$ for right-folded $\oplus$ we have $F{\cdot}x = [\,]\otimes F{\cdot}x$, and we derive, for the two cases $\neg b{\cdot}x$ and $b{\cdot}x$ seperately:

$$
\begin{array}{ll}
 & s \otimes F{\cdot}x \\
= & \quad \{ \text{ definition of } F\text{, case } \neg b{\cdot}x \ \} \\
 & s \otimes f{\cdot}x \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & s \otimes F{\cdot}x \\
= & \quad \{ \text{ definition of } F\text{, case } b{\cdot}x \ \} \\
 & s \otimes (h{\cdot}x \oplus F{\cdot}(g{\cdot}x)) \\
= & \quad \{ \otimes \} \\
 & (s \triangleleft h{\cdot}x) \otimes F{\cdot}(g{\cdot}x) \quad .
\end{array}
$$

Inspired by this we now introduce a function G with specification:

$$G{\cdot}s{\cdot}x \ = \ s \otimes F{\cdot}x \quad , \text{ for all } s, x \quad .$$

Then we have $F{\cdot}x = G{\cdot}[\,]{\cdot}x$, so G is a generalization of F and, as a result of the above derivation, we obtain the following theorem.

second tail-recursion theorem:

$$
\begin{array}{lrcl}
\text{if:} & F{\cdot}x & = & \mathsf{if} \ \neg b{\cdot}x \ \rightarrow \ f{\cdot}x \\
 & & & [\!] \ \ b{\cdot}x \ \rightarrow \ h{\cdot}x \oplus F{\cdot}(g{\cdot}x) \\
 & & & \mathsf{fi} \\
 & & & \\
\text{then:} & F{\cdot}x & = & G{\cdot}[\,]{\cdot}x \\
 & & & \\
\text{where:} & G{\cdot}s{\cdot}x & = & \mathsf{if} \ \neg b{\cdot}x \ \rightarrow \ s \otimes f{\cdot}x \\
 & & & [\!] \ \ b{\cdot}x \ \rightarrow \ G{\cdot}(s \triangleleft h{\cdot}x){\cdot}(g{\cdot}x) \\
 & & & \mathsf{fi} \\
\text{and:} & & & \\
 & [\,] \otimes v & = & v \\
\& & (s \triangleleft y) \otimes v & = & s \otimes (y \oplus v)
\end{array}
$$

By this theorem every linearly recursive function F can be rewritten as the composition of two tail-recursive functions, G and $\otimes$. The "interface" between these two functions is the list parameter s which can be interpreted as the "stack" that is traditionally used to implement recursion.

Notice again, however, that we have not used this operational connotation to invent the theorem; again, we have derived it by just letting the symbols do the work. This also shows that the concept of "the stack" requires no invention: it just emerges as the result of a generalization.

* * *

The above definitions can be easily implemented as a sequential program for the computation of $F\cdot x0$, for some given value $x0$. In the following program, for example, list parameter s is represented by an array variable c and an integer p –for the length of s–, according to the representation invariant:

Q: $\quad p = \#s \ \wedge \ (\forall i : 0 \le i < p : s\cdot i = c[i])$

Recall that a (finite) list is completely determined by its elements, so to represent a list it suffices to represent the list's length and its elements. Here predicate Q expresses that p is the length of s and that the first p elements of array c hold the elements of s.

sequential program iiia:

$$
\begin{array}{ll}
 & p, x := 0, x0 \\
; & \{ \text{ invariant: } \ F\cdot x0 = G\cdot s\cdot x \ \wedge \ Q \ \} \\
 & \text{do } \ b\cdot x \ \rightarrow \ c[p] := h\cdot x \ ; \ p := p{+}1 \ ; \ x := g\cdot x \ \text{ od} \\
; & \{ \ F\cdot x0 = s \otimes f\cdot x \ \} \\
 & v := f\cdot x \\
; & \{ \text{ invariant: } \ F\cdot x0 = s \otimes v \ \wedge \ Q \ \} \\
 & \text{do } \ p{\neq}0 \ \rightarrow \ p := p{-}1 \ ; \ v := c[p] \oplus v \ \text{ od} \\
 & \{ \ F\cdot x0 = v \ \}
\end{array}
$$

□

In the first repetition of this program, values $h\cdot x$ are stored in array c, with $x0$, $g\cdot x0$, $g\cdot(g\cdot x0)$, $\cdots$ for the successive values of x. In the second repetition these values are processed, in the statement $v := c[p] \oplus v$.

As an alternative, it is possible to move the application of function h to the second repetition; that is, $c[p] := h\cdot x$ is replaced by $c[p] := x$ and $v := c[p] \oplus v$ is replaced by $v := h\cdot(c[p]) \oplus v$. This yields the following, equivalent –see below–, program.

sequential program iiib:

$$
\begin{array}{ll}
 & p, x := 0, x0 \\
; & \{ \text{ invariant: } F\cdot x0 = G\cdot s\cdot x \ \wedge \ Q \ \} \\
 & \text{do } b\cdot x \ \rightarrow \ c[p] := x \ ; \ p := p{+}1 \ ; \ x := g\cdot x \ \text{ od} \\
; & \{ \ F\cdot x0 = s \otimes f\cdot x \ \} \\
 & v := f\cdot x \\
; & \{ \text{ invariant: } F\cdot x0 = s \otimes v \ \wedge \ Q \ \} \\
 & \text{do } p{\neq}0 \ \rightarrow \ p := p{-}1 \ ; \ v := h\cdot(c[p]) \oplus v \ \text{ od} \\
 & \{ \ F\cdot x0 = v \ \}
\end{array}
$$

□

How do we convince ourselves of the correctness of this program? The simplest way is not to consider this program as being obtained from the former one, but to consider it as the implementation of a functional program, as follows.

We eliminate function h from the definition of function F, by redefining operator $\oplus$, calling it $\oplus'$:

$$x \oplus' v \ = \ h\cdot x \oplus v \quad \text{, for all } x, v \quad .$$

In terms of $\oplus'$ the definition of F can now be rewritten as:

$$
\begin{array}{lll}
F\cdot x & = & \text{if } \neg b\cdot x \ \rightarrow \ f\cdot x \\
 & & [] \quad b\cdot x \ \rightarrow \ x \oplus' F\cdot(g\cdot x) \\
 & & \text{fi}
\end{array}
$$

If we now apply the second tail-recursion theorem to this definition of F, that is with $\oplus'$ instead of $\oplus$ and with the identity function for h, then the sequential implementation of functions G and $\otimes$ resulting from the theorem, directly lead to program iiib.

The moral of this story is that program transformations preferably are carried out in the world of functional programs, thus postponing the transition to a sequential implementation as long as possible.

The transformation from program iiia into program iiib may improve the space efficiency of the program, namely if the values of type X require less storage space than the values of type Y –the type of $h\cdot x$–. In addition the *order* in which successive values of h are computed is *reversed*; this may be of interest if we wish to improve the program's efficiency by exploiting recurrence relations of function h.

8.9 Tail Recursion for Functions on Lists

We consider a function F, of type $\mathcal{L}_*(B) \to V$, for some element type B and type V, and defined recursively as follows:

$$\begin{array}{lll} & F\cdot[\,] & = & v0 \\ \& & F\cdot(b \triangleright t) & = & b \oplus F\cdot t \end{array}$$

Here, $v0$ is a constant of type V and $\oplus$ is a binary operator of type $B \times V \to V$. Function F is the prototype of a linearly recursive function on a domain of lists.

With $\otimes$ for right-folded $\oplus$, we now have:

$$\begin{array}{lll} F\cdot t & = & [\,] \otimes F\cdot t \quad , \text{ and:} \\ F\cdot(b \triangleright t) & = & [b] \otimes F\cdot t \quad . \end{array}$$

Because $t = [\,] +\!\!+ t$ and $b \triangleright t = [b] +\!\!+ t$ we can rewrite these properties as:

$$\begin{array}{lll} F\cdot([\,] +\!\!+ t) & = & [\,] \otimes F\cdot t \quad , \text{ and:} \\ F\cdot([b] +\!\!+ t) & = & [b] \otimes F\cdot t \quad . \end{array}$$

From these relations we obtain the impression that F might satisfy the more general property, for all lists s, t:

$$(5) \qquad F\cdot(s +\!\!+ t) \;=\; s \otimes F\cdot t \quad .$$

This property holds indeed, as can be shown by a simple proof by Mathematical Induction on the size of s.

From (5), with $t := [\,]$, and using $s +\!\!+ [\,] = s$ and $F\cdot[\,] = v0$, we obtain:

$$(6) \qquad F\cdot s \;=\; s \otimes v0 \quad .$$

Because neither the expression $s \otimes v0$ nor the definitions for $\otimes$ contain F, formula (6) can be used as an alternative definition for F.

As we have seen, $\otimes$ admits several recursive definitions, one of which is:

$$\begin{array}{lll} & [\,] \otimes v & = & v \\ \& & (b \triangleright t) \otimes v & = & b \oplus (t \otimes v) \end{array}$$

This amounts to just a slight generalization – constant $v0$ replaced by parameter v – of F's definition. Another pair of definitions of $\otimes$, however, is tail-recursive:

$$
\begin{array}{rrcl}
 & [\,] \otimes v & = & v \\
\& & (s \lhd c) \otimes v & = & s \otimes (c \oplus v)
\end{array}
$$

Together with (6) this forms a tail-recursive definition for F.

A minor complication may be that, as the occurrence of $\lhd$, instead of $\rhd$, shows, the elements of the list are interpreted in reverse order. If so desired, this can be remedied by means of the rev-trick, that is, by introduction of a new function G, with specification:

$$G \cdot v \cdot t \;=\; rev \cdot t \otimes v \quad \text{, for all } v, t \quad .$$

Thus, the list reversal is made explicit. This standard transformation yields the following solution:

tail-recursion theorem (i) for lists:

if:

$$
\begin{array}{rrcl}
 & F \cdot [\,] & = & v0 \\
\& & F \cdot (b \rhd t) & = & b \oplus F \cdot t
\end{array}
$$

then: $F \cdot (rev \cdot t) \;=\; G \cdot v0 \cdot t$

hence also: $F \cdot s \;=\; G \cdot v0 \cdot (rev \cdot s)$

where:

$$
\begin{array}{rrcl}
 & G \cdot v \cdot [\,] & = & v \\
\& & G \cdot v \cdot (b \rhd t) & = & G \cdot (b \oplus v) \cdot t
\end{array}
$$

* * *

The above theorem contains occurrences of function rev. If so desired, these occurrences may be fused with the functions in which they occur, by applying the properties of function rev. For example, this is one of the possible definitions for rev, using recursion based on $\lhd$:

$$
\begin{array}{rrcl}
 & rev \cdot [\,] & = & [\,] \\
\& & rev \cdot (t \lhd b) & = & b \rhd rev \cdot t
\end{array}
$$

Now we can introduce a function Frv, say, with this specification, for list t:

$$Frv \cdot t \;=\; F \cdot (rev \cdot t) \quad .$$

Using the given definitions for F and rev a recursive definition for Frv is easily derived, by induction on t; this derivation is entirely of the type "nothing-else-you-can-do":

$$\begin{array}{ll}
 & Frv \cdot [\,] \\
= & \quad \{ \text{ specification of } Frv \text{ } \} \\
 & F \cdot (rev \cdot [\,]) \\
= & \quad \{ \text{ definition of } rev \text{ } \} \\
 & F \cdot [\,] \\
= & \quad \{ \text{ definition of } F \text{ } \} \\
 & v0 \quad ,
\end{array}$$

and:

$$\begin{array}{ll}
 & Frv \cdot (t \lhd b) \\
= & \quad \{ \text{ specification of } Frv \text{ } \} \\
 & F \cdot (rev \cdot (t \lhd b)) \\
= & \quad \{ \text{ definition of } rev \text{ } \} \\
 & F \cdot (b \rhd rev \cdot t) \\
= & \quad \{ \text{ definition of } F \text{ } \} \\
 & b \oplus F \cdot (rev \cdot t) \\
= & \quad \{ \text{ specification of } Frv, \text{ by Induction Hypothesis } \} \\
 & b \oplus Frv \cdot t \quad .
\end{array}$$

Thus, we obtain this recursive definition for Frv, based on the $\lhd$-structure of lists:

$$\begin{array}{lll}
 & Frv \cdot [\,] & = \; v0 \\
\& & Frv \cdot (t \lhd b) & = \; b \oplus Frv \cdot t
\end{array}$$

Substituting this in the tail-recursion theorem, and subsequently renaming Frv into F again, we obtain this, equivalent, version of the theorem, in which now function rev is implicitly present, in the transition from $\lhd$-lists to $\rhd$-lists.

tail-recursion theorem (ii) for lists:

$$
\begin{array}{lrcl}
\text{if:} & & & \\
 & F\cdot[\,] & = & v0 \\
\& & F\cdot(t \lhd b) & = & b \oplus F\cdot t \\
 & & & \\
\text{then:} & F\cdot t & = & G\cdot v0\cdot t \\
 & & & \\
\text{where:} & & & \\
 & G\cdot v\cdot[\,] & = & v \\
\& & G\cdot v\cdot(b \rhd t) & = & G\cdot(b \oplus v)\cdot t
\end{array}
$$

This version of the theorem expresses F, on $\lhd$-lists, in terms of G, on $\rhd$-lists. Obviously, yet another version of this theorem can be formulated, expressing F, on $\rhd$-lists, in terms of G, on $\lhd$-lists.

8.9.0 example: summing a list

We consider a function F mapping integer lists to their sums; that is, F's specification is, for integer list t of length n:

$$F\cdot t \;=\; (\Sigma i : 0 \le i < n : t\cdot i) \quad .$$

A recursive definition for F is:

$$
\begin{array}{lrcl}
 & F\cdot[\,] & = & 0 \\
\& & F\cdot(b \rhd t) & = & b + F\cdot t
\end{array}
$$

Now there are two ways to transform this definition into a tail recursive one: firstly, by observing that addition has an identity element and is associative we can apply the first tail-recursion theorem; secondly, we can apply the tail-recursion theorem for lists.

The latter yields $F\cdot(rev\cdot t) = G\cdot 0\cdot t$, where in this example G is to be defined by:

$$
\begin{array}{lrcl}
 & G\cdot v\cdot[\,] & = & v \\
\& & G\cdot v\cdot(b \rhd t) & = & G\cdot(b+v)\cdot t
\end{array}
$$

Because addition is both associative and commutative we also have that summation of the elements of a list is invariant under list reversal; so, we have $F\cdot(rev\cdot t) = F\cdot t$ and, hence, we also have $F\cdot t = G\cdot 0\cdot t$.

Thus, the result of the transformation is exactly the same as the result obtained by application of the first tail-recursion theorem.

8.10 Multiple Recursion: an example

Expressions are linear strings of symbols representing tree structures. Evaluators are mechanisms for computing the "values" of such trees and are usually designed to take expressions for their inputs, instead of trees. This is a matter of efficiency: executing a linear sequence of instructions may take less time than performing a tree walk, and this linear sequence may take less space for its storage than the tree it represents.

From a definition of trees as a starting point, an evaluator can be designed, firstly, by *choosing* a linear representation and, next, by deriving the evaluator. This choice is not trivial, though, because trees can be represented linearly in many different ways, such as by prefix, infix, or postfix codes: at the outset, it is not at all clear which choice will lead to an efficient design.

Here we illustrate a design approach in which the linear representation just *emerges* as a by-product of the design process. The example is about simple expressions, which either are *primitive* terms – like constants, variables, or function applications –, or expressions *composed* from other expressions by means of binary operators.

8.10.0 trees and their values

In what follows, dummy b has type B and dummy c has type C, for some given and *disjoint* types B and C, so we have $(\forall b, c :: b \neq c)$. The datatype T of trees is defined (recursively) as *smallest* set satisfying:

$$\begin{array}{lll} \langle b \rangle & \in T & \text{, for all } b \\ \langle s, c, t \rangle & \in T & \text{, for all } c \text{ and } s, t \in T \ . \end{array}$$

Next, we assume that, for some given type M, functions f and g have been given, with the following types:

$$\begin{array}{lll} f & \in & B \to M \\ g & \in & C \to M \to M \to M \end{array}$$

Type M is the type of the values of trees, as is reflected by the following definition of function $V \in T \to M$, with the intention that $V \cdot s$ be the value of tree s:

$$\begin{array}{lcl} V \cdot \langle b \rangle & = & f \cdot b \\ V \cdot \langle s, c, t \rangle & = & g \cdot c \cdot (V \cdot s) \cdot (V \cdot t) \end{array}$$

8.10.1 an evaluator

Assuming that we know how to implement f and g, we are interested in implementations of V with such a fine grain of detail that the result can be considered as code for a Von Neumann type of machine. In particular we wish to eliminate the recursion from V's definition, although we may equally well say that we wish to make the implementation of this recursion explicit.

A standard technique to eliminate recursion is to transform it into tail-recursion, which can be implemented by simple iteration. As a first step, we try to reduce the *two* recursive occurrences of V in its definition to a *single* one, thus making the definition linearly recursive: After all, we know how to deal with linear recursion. The main design decision now is that we collect *many trees* into a *single list of trees*, that is, we study $V \bullet ss$, for ss of type $\mathcal{L}_*(T)$, for two reasons. First, it is a generalization, because $[\, V \cdot s \,] = V \bullet [\, s \,]$, and, second, $V \cdot s$ and $V \cdot t$ are the elements of the list $V \bullet [\, s, t \,]$; thus, we can combine the two recursive applications of V into a single one.

As always we have $V \bullet [\,] = [\,]$; furthermore, in compliance with the case analysis in V's definition, we derive:

$$\begin{array}{ll} & V \bullet (\langle b \rangle \triangleright ss) \\ = & \quad \{ \ \bullet \ \} \\ & V \cdot \langle b \rangle \ \triangleright \ V \bullet ss \\ = & \quad \{ \ V \ \} \\ & f \cdot b \ \triangleright \ V \bullet ss \\ = & \quad \{ \text{ introduction of } \oplus \text{ (motivation follows, see below) } \} \\ & b \oplus V \bullet ss \quad , \end{array}$$

and:

$$\begin{array}{ll} & V \bullet (\langle s, c, t \rangle \triangleright ss) \\ = & \quad \{ \ \bullet \ \} \end{array}$$

$$
\begin{array}{ll}
 & V\cdot\langle s,c,t\rangle \triangleright V\bullet ss \\
= & \quad \{\ V\ \} \\
 & g\cdot c\cdot(V\cdot s)\cdot(V\cdot t) \triangleright V\bullet ss \\
= & \quad \{\ \text{eureka! introduction of } \oplus \text{ (see below)}\ \} \\
 & c \oplus (V\cdot s \triangleright V\cdot t \triangleright V\bullet ss) \\
= & \quad \{\ \bullet\ \text{(twice)}\ \} \\
 & c \oplus (V\bullet(s\triangleright t\triangleright ss))\quad .
\end{array}
$$

Here we have introduced an operator $\oplus$ defined by, for all $m,n\in M$ and $ms\in\mathcal{L}_*(M)$:

$$
\begin{array}{lcl}
b \oplus ms & = & f\cdot b \triangleright ms \\
c \oplus (m\triangleright n\triangleright ms) & = & g\cdot c\cdot m\cdot n \triangleright ms
\end{array}
$$

This is not at as far-fetched as it may seem: the only way to stay within the pattern – expressions of the shape $V\bullet(\cdots)$ – is to collect $V\cdot s$, $V\cdot t$, and $V\bullet ss$ into $V\bullet(s\triangleright t\triangleright ss)$. Phrased differently, this step exploits the observation that the expression $g\cdot c\cdot(V\cdot s)\cdot(V\cdot t) \triangleright V\bullet ss$ is a function of its subexpressions c, $V\cdot s$, $V\cdot t$, and $V\bullet ss$.

Notice that the overloading of $\oplus$ is harmless here: because the types of b and c are disjoint the two cases in its definition can be distinguished. Thus we obtain as a linearly recursive definition for function $(V\bullet)$:

$$
\begin{array}{lcl}
V\bullet\,[\,] & = & [\,] \\
V\bullet(\langle b\rangle \triangleright ss) & = & b \oplus (V\bullet ss) \\
V\bullet(\langle s,c,t\rangle \triangleright ss) & = & c \oplus (V\bullet(s\triangleright t\triangleright ss))
\end{array}
$$

Now we apply the second tail-recursion theorem to transform this linear recursion into tail recursion. With $\otimes$ for right-folded $\oplus$ we obtain, in this case, as definition for $\otimes$:

$$
\begin{array}{lcl}
[\,] \otimes ms & = & ms \\
(x\triangleleft b) \otimes ms & = & x \otimes (f\cdot b \triangleright ms) \\
(x\triangleleft c) \otimes (m\triangleright n\triangleright ms) & = & x \otimes (g\cdot c\cdot m\cdot n \triangleright ms)
\end{array}
$$

The tail-recursion theorem now tells us that $(V\bullet)$ can be defined in terms of a function G as follows:

$$
V\bullet ss \quad = \quad G\cdot[\,]\cdot ss \quad ,
$$

where function G, of type $\mathcal{L}_*(B{\cup}C) \to \mathcal{L}_*(T) \to \mathcal{L}_*(M)$, has as specification:

$$G{\cdot}x{\cdot}ss \quad = \quad x \otimes V{\bullet}ss \quad .$$

Moreover, the theorem yields as definition for G:

$$\begin{array}{lcl} G{\cdot}x{\cdot}[\,] & = & x \otimes [\,] \\ G{\cdot}x{\cdot}(\langle b\rangle \triangleright ss) & = & G{\cdot}(x{\triangleleft}b){\cdot}ss \\ G{\cdot}x{\cdot}(\langle s,c,t\rangle \triangleright ss) & = & G{\cdot}(x{\triangleleft}c){\cdot}(s{\triangleright}t{\triangleright}ss) \end{array}$$

As a special case we obtain as solution for our original problem:

$$[\,V{\cdot}s\,] \;=\; G{\cdot}[\,]{\cdot}[\,s\,] \;\text{, hence: } V{\cdot}s \;=\; G{\cdot}[\,]{\cdot}[\,s\,]{\cdot}0 \;\; .$$

$$* \qquad\qquad * \qquad\qquad *$$

According to the rule for tail-fusion – cf. Section 8.1 – we can view G as the composition of $(\otimes\,[\,])$ and a function F, say:

$$G{\cdot}x{\cdot}ss \quad = \quad F{\cdot}x{\cdot}ss \otimes [\,] \quad ,$$

where a definition for F is obtained from G's definition by simply omitting $\otimes\,[\,]$. Summarizing, we obtain the following set of definitions:

$$\boxed{\begin{array}{lcl} [\,V{\cdot}s\,] & = & F{\cdot}[\,]{\cdot}[\,s\,] \otimes [\,] \\ \\ F{\cdot}x{\cdot}[\,] & = & x \\ F{\cdot}x{\cdot}(\langle b\rangle \triangleright ss) & = & F{\cdot}(x{\triangleleft}b){\cdot}ss \\ F{\cdot}x{\cdot}(\langle s,c,t\rangle \triangleright ss) & = & F{\cdot}(x{\triangleleft}c){\cdot}(s{\triangleright}t{\triangleright}ss) \\ \\ [\,] \otimes ms & = & ms \\ (x{\triangleleft}b) \otimes ms & = & x \otimes (f{\cdot}b \triangleright ms) \\ (x{\triangleleft}c) \otimes (m{\triangleright}n{\triangleright}ms) & = & x \otimes (g{\cdot}c{\cdot}m{\cdot}n \triangleright ms) \end{array}}$$

These definitions admit a nice operational interpretation. The value of $[\,V{\cdot}s\,]$ can be computed in two phases: first, $F{\cdot}[\,]{\cdot}[\,s\,]$ is computed, which yields some list x (of type $\mathcal{L}_*(B{\cup}C)$), and, second, $x{\otimes}[\,]$ is evaluated. The

first phase, and the resulting list x, are independent of f and g: it can be considered as a purely *syntactic* transformation, that is, x is just a linear representation of tree s. Actually, x is the postfix-code representation of s when, conforming with the snoc operations in the definition of $\otimes$, we "read x from right to left". (For example, $F\cdot[\,]\cdot[\langle\langle b_0, c_1, b_1\rangle, c_0, \langle b_2\rangle\rangle] \;=\; [c_0, c_1, b_0, b_1, b_2]$.)

The first phase can be performed "at compile time" and is known as "code generation". The second phase then becomes "program execution" where the list computed by the first phase is the program to be executed. A value b in this list can be viewed as an instruction to "push value $f\cdot b$ onto the stack" whereas a value c can be viewed as an instruction to "apply operation $g\cdot c$ to the top two elements of the stack and replace them by the result".

* * *

The above is independent of the actual choice of M, f, and g. With $M = \mathsf{Int}$ and an appropriate choice for f and g we obtain an evaluator for integer expressions. But we may also choose:

$$\begin{array}{lcl} M & = & T \\ f\cdot b & = & \langle b\rangle \\ g\cdot c\cdot s\cdot t & = & \langle s, c, t\rangle \end{array}$$

Now V becomes the identity function on T and we obtain:

$$[\,s\,] \;=\; F\cdot[\,]\cdot[\,s\,] \otimes [\,] \quad ,$$

which means that $(\otimes[\,])$ reconstructs $[\,s\,]$ from $F\cdot[\,]\cdot[\,s\,]$: now $(\otimes[\,])$ is a *(bottom-up) parser* for postfix-code expressions. For this case we obtain:

$$\begin{array}{rcl} [\,] \otimes ss & = & ss \\ (x \triangleleft b) \otimes ss & = & x \otimes (\langle b\rangle \triangleright ss) \\ (x \triangleleft c) \otimes (s \triangleright t \triangleright ss) & = & x \otimes (\langle s, c, t\rangle \triangleright ss) \end{array}$$

Generally, $ss = F\cdot[\,]\cdot ss \otimes [\,]$, so $(\otimes[\,])$ is the inverse of $F\cdot[\,]$: parsing is the inverse operation of representing a tree by a list.

8.11 More examples

8.11.0 representation conversion revisited

In Section 7.3.4 we have constructed a functional program for the conversion of binary numbers into ternary numbers. This involved a function *c23* mapping binary lists to ternary lists and an auxiliary function f. We repeat the definitions of these functions here:

$$
\begin{array}{lll}
 & c23\cdot[\,] & = & [\,] \\
\& & c23\cdot(s \triangleleft b) & = & f\cdot b\cdot(c23\cdot s) \\
 & & & \\
 & f\cdot b\cdot[\,] & = & [b] \\
\& & f\cdot b\cdot(t \triangleleft c) & = & f\cdot(h\,\mathsf{div}\,3)\cdot t \triangleleft h\,\mathsf{mod}\,3 \\
 & & & \mathsf{whr}\ \ h = 2*c+b\ \ \mathsf{end}
\end{array}
$$

We now apply the tail-recursion theorems to transform these recursive definitions into a sequential program for the conversion of binary into ternary numbers.

We start with $c23$. This is a function on lists and the recursion pattern of its definition matches the tail-recursion (ii) theorem for lists. Here function f corresponds to binary operator $\oplus$ in the theorem: we may define $\oplus$ by, for binary digit b and ternary list t:

$$b \oplus t \;=\; f\cdot b\cdot t \quad .$$

The tail-recursion theorem for lists now yields that:

$$c23\cdot s \;=\; G\cdot[\,]\cdot s \quad ,$$

provided we define function G by:

$$
\begin{array}{lll}
 & G\cdot t\cdot[\,] & = & t \\
\& & G\cdot t\cdot(b \triangleright s) & = & G\cdot(f\cdot b\cdot t)\cdot s
\end{array}
$$

From this tail-recursive definition we obtain the following sequential program for the computation of $F\cdot s0$, for some given list $s0$, of length N. In the following program the list parameter s is represented by an array variable c and an integer p –for the length of s–, according to the representation invariant:

$$P0: \quad \#s = N-p \;\wedge\; (\forall i : 0 \le i < N-p : s\cdot i = c[p+i]) \quad .$$

That is, the elemtents of s are stored in array segment $c[p\,..\,N)$. We assume that the initial value $c[0\,..\,N)$ contains the elements of $s0$. Variable t is assumed to be a list; its representation and the implementation of $t := f\cdot b\cdot t$ will be taken care of in the next step.

$$
\begin{array}{ll}
 & \{\ p = \#s0\ \wedge\ (\forall i : 0 \le i < p : s0 \cdot i = c[i])\ \} \\
 & t, p := [\,], 0 \\
; & \{\ \mathsf{invariant}:\ 0 \le p \le N\ \wedge\ P0\ \wedge\ c23 \cdot s0 = G \cdot t \cdot s\ \} \\
 & \mathsf{do}\ p \ne N\ \rightarrow\ b := c[p]\ ;\ p := p+1 \\
 & \qquad\qquad ;\ t := f \cdot b \cdot t \\
 & \mathsf{od} \\
 & \{\ t\ =\ c23 \cdot s0\ \}
\end{array}
$$

Next we consider the implementation of function f, namely the implementation of the assignment $t := f \cdot b \cdot t$.

Observing that $(\triangleleft\ h\,\mathsf{mod}\,3)$ is equivalent to $(+\!\!+\ [h\,\mathsf{mod}\,3])$, with the non-associative operator $\triangleleft$ replaced by the associative $+\!\!+$, we prepare for application of the first tail-recursion theorem, by rewriting f's definition:

$$
\begin{array}{lll}
 & f \cdot b \cdot [\,] & = & [b] \\
\& & f \cdot b \cdot (t \triangleleft c) & = & f \cdot (h\,\mathsf{div}\,3) \cdot t\ +\!\!+\ [h\,\mathsf{mod}\,3] \\
 & & & \mathsf{whr}\ \ h\ =\ 2 * c + b\ \ \mathsf{end}
\end{array}
$$

Now we can easily generalize f to a function g, with this specification:

$$
g \cdot s \cdot b \cdot t\ \ =\ \ f \cdot b \cdot t\ +\!\!+\ s\ \ .
$$

So, we have $f \cdot b \cdot t\ =\ g \cdot [\,] \cdot b \cdot t$, and:

$$
\begin{array}{lll}
 & g \cdot s \cdot b \cdot [\,] & = & b \triangleright s \\
\& & g \cdot s \cdot b \cdot (t \triangleleft c) & = & g \cdot (h\mathsf{mod}3 \triangleright s) \cdot (h\mathsf{div}3) \cdot t \\
 & & & \mathsf{whr}\ \ h\ =\ 2 * c + b\ \ \mathsf{end}
\end{array}
$$

8.11.1 Ackermann's function

We consider a binary operator $\oplus$, defined recursively in terms of given expressions $B_0, B_1, B_2, C_0, C_1, C_2, V_0, V_1$ and V_2, as follows:

$$
\begin{array}{lll}
b \oplus v & = & \mathsf{if}\ \ B_0\ \rightarrow\ V_0 \\
 & & [\!]\ \ B_1\ \rightarrow\ C_0 \oplus V_1 \\
 & & [\!]\ \ B_2\ \rightarrow\ C_2 \oplus (C_1 \oplus V_2) \\
 & & \mathsf{fi}
\end{array}
$$

Obviously, the parameters b and v may occur in the given expressions but, for the sake brevity, this dependence is left implicit.

In order that this definition is meaningful indeed, the recursion must be well-founded. For this purpose we postulate the existence of a well-founded relation $<$, with the following properties, on the set of all (relevant) pairs $\langle b, v \rangle$, for all b, v:

$$\begin{array}{lll} B_1 & \Rightarrow & \langle C_0, V_1 \rangle < \langle b, v \rangle \\ B_2 & \Rightarrow & \langle C_1, V_2 \rangle < \langle b, v \rangle \ \wedge \ \langle C_2, C_1 \oplus V_2 \rangle < \langle b, v \rangle \end{array}$$

Using these properties and the well-foundedness of relation $<$, we can now prove properties of $\oplus$ by Mathematical Induction.

The first case in $\oplus$'s definition is the base of the recursion, the second case is tail recursive, and the third case is a double recursion that is *nested*: one recursive application of $\oplus$ has the result of the other recursive application as one of its arguments.

Our goal in this example is to transform this definition into a completely tail-recursive one; this requires the elimination of the nested recursion.

* * *

The shapes of the right-hand side expressions in $\oplus$'s definition indicate that we should study right-folded $\oplus$, which – as usual now – we denote by $\otimes$. From Section 8.7 we recall a definition for $\otimes$:

$$\begin{array}{rcl} [\,] \otimes v & = & v \\ (s \triangleleft b) \otimes v & = & s \otimes (b \oplus v) \end{array}$$

These properties define $\otimes$ in terms of $\oplus$. Conversely, as before, $\oplus$ can be defined in terms of $\otimes$ by:

$$b \oplus v \ = \ [b] \otimes v \quad .$$

So, our goal now is to derive a definition for $\otimes$ in which $\oplus$ does not occur anymore. If we succeed and if the definition thus obtained is tail-recursive, we have reached the goal we started with.

As usual, the derivation follows the case analysis in the definition of $\oplus$; independently of $\oplus$ we have:

$$[\,] \otimes v \ = \ v \quad ,$$

and, furthermore, we derive:

$$\begin{array}{ll} & (s \triangleleft b) \otimes v \\ = & \quad \{ \text{ definition of } \otimes \text{, here serving as specification } \} \\ & s \otimes (b \oplus v) \\ = & \quad \{ \text{ definition of } \oplus \text{, case } B_0 \ \} \end{array}$$

$$s \otimes V_0 \quad ,$$

and:

$$\begin{array}{ll}
 & (s \lhd b) \otimes v \\
= & \quad \{ \text{ definition of } \otimes \text{, again serving as specification } \} \\
 & s \otimes (b \oplus v) \\
= & \quad \{ \text{ definition of } \oplus \text{, case } B_1 \ \} \\
 & s \otimes (C_0 \oplus V_1) \\
= & \quad \{ \text{ definition of } \otimes \text{, by Induction Hypothesis } \} \\
 & (s \lhd C_0) \ \otimes \ V_1 \quad ,
\end{array}$$

and:

$$\begin{array}{ll}
 & (s \lhd b) \otimes v \\
= & \quad \{ \text{ definition of } \otimes \text{, again serving as specification } \} \\
 & s \otimes (b \oplus v) \\
= & \quad \{ \text{ definition of } \oplus \text{, case } B_2 \ \} \\
 & s \otimes (C_2 \oplus (C_1 \oplus V_2)) \\
= & \quad \{ \text{ definition of } \otimes \text{, by Induction Hypothesis } \} \\
 & (s \lhd C_2) \ \otimes \ (C_1 \oplus V_2) \\
= & \quad \{ \text{ definition of } \otimes \text{, by Induction Hypothesis } \} \\
 & (s \lhd C_2 \lhd C_1) \ \otimes \ V_2 \quad .
\end{array}$$

Collecting these results we obtain the following tail-recursive definition for $\otimes$:

$$\begin{array}{rrcl}
 & [\,] \otimes v & = & v \\
\& & (s \lhd b) \otimes v & = & \mathsf{if} \ \ B_0 \ \rightarrow \ s \otimes V_0 \\
 & & & [\,] \ \ B_1 \ \rightarrow \ (s \lhd C_0) \ \otimes \ V_1 \\
 & & & [\,] \ \ B_2 \ \rightarrow \ (s \lhd C_2 \lhd C_1) \ \otimes \ V_2 \\
 & & & \mathsf{fi}
\end{array}$$

This example shows that (the implementation of) nested recursion poses no particular problems. As in the previous examples, list s emerges in the transition from $\oplus$ to its right-folded $\otimes$, and, as before, list s can be viewed as "the stack".

* * *

We apply the above to a simplified version of what is known as "Ackermann's function". For the complete version of this function we refer to the exercises.

Function A, of type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$, is defined recursively by:

$$
\begin{array}{lcll}
A \cdot i \cdot j & = & \mathsf{if}\ \ i = 0 & \to\ \ j+1 \\
 & & [\!]\ \ i > 0 \wedge j = 0 & \to\ \ A \cdot (i-1) \cdot 1 \\
 & & [\!]\ \ i > 0 \wedge j > 0 & \to\ \ A \cdot (i-1) \cdot (A \cdot i \cdot (j-1)) \\
 & & \mathsf{fi} &
\end{array}
$$

Because the lexicographic order on pairs of naturals is well-founded, this is a well-founded definition. Within this order we have $\langle i-1\,, j \rangle < \langle i, j \rangle$, $\langle i\,, j-1 \rangle < \langle i, j \rangle$, and $\langle\, i-1\,, A \cdot i \cdot (j-1)\,\rangle < \langle i, j \rangle$.

The recursion pattern of this definition is of the kind described above: just take $i \oplus j \;=\; A \cdot i \cdot j$. By application of the result derived above, we obtain that:

$$
A \cdot i \cdot j \;\;=\;\; [\,i\,] \otimes j \quad ,
$$

where, for this example, $\otimes$ is defined as follows:

$$
\begin{array}{lrcll}
 & [\,] \otimes j & = & j & \\
\& & (s \lhd i) \otimes j & = & \mathsf{if}\ \ i = 0 & \to\ \ s \otimes (j+1) \\
 & & & [\!]\ \ i > 0 \wedge j = 0 & \to\ \ (s \lhd (i-1)) \otimes 1 \\
 & & & [\!]\ \ i > 0 \wedge j > 0 & \to\ \ (s \lhd (i-1) \lhd i) \otimes (j-1) \\
 & & & \mathsf{fi} &
\end{array}
$$

* * *

Because $A \cdot i \cdot j \;=\; [\,i\,] \otimes j$ and because $[\,0\,] \otimes j \;=\; j+1$, we can easily restrict the use of $s \otimes j$ to the cases where list s is *non-empty*. For such non-empty lists we perform a small change of representation, namely by representing the list's last element as a separate parameter. To this end we introduce a new function G, say, with this specification, for all lists s and natural i, j:

$$
G \cdot s \cdot i \cdot j \;\;=\;\; (s \lhd i) \otimes j \quad .
$$

So, with the new function parameters s and i together represent list $s \lhd i$. In terms of G function A can be defined by:

$$
A \cdot i \cdot j \;\;=\;\; G \cdot [\,] \cdot i \cdot j \quad ,
$$

and by means of straightforward calculations we obtain as definition for G:

$$
\begin{array}{lll}
 & G\cdot[\,]\cdot 0\cdot j & = \; j+1 \\
\& & G\cdot(s\triangleleft i)\cdot 0\cdot j & = \; G\cdot s\cdot i\cdot(j{+}1) \\
\& & G\cdot s\cdot(i{+}1)\cdot 0 & = \; G\cdot s\cdot i\cdot 1 \\
\& & G\cdot s\cdot(i{+}1)\cdot(j{+}1) & = \; G\cdot(s\triangleleft i)\cdot(i{+}1)\cdot j
\end{array}
$$

Just for the fun of it, we conclude this example with a sequential program for the computation of $A\cdot M\cdot N$, for some given naturals M and N. In this program list s is represented by an array c and a natural variable p, in a way we have seen earlier, with this representation invariant:

$$p = \#s \;\wedge\; (\forall i : 0\le i<p : s\cdot i = c[i]) \quad .$$

sequential program v:

$$
\begin{array}{ll}
 & \{\; 0\le M \;\wedge\; 0\le N \;\} \\
 & p, m, n := 0, M, N \\
; & \{\; \mathsf{invariant}: 0\le m \;\wedge\; 0\le n \;\wedge\; A\cdot M\cdot N = G\cdot s\cdot m\cdot n \;\} \\
 & \mathsf{do}\;\; m=0 \wedge p>0 \;\rightarrow\; p := p-1 \;;\; m, n := c[p], n+1 \\
 & [\,]\;\; m>0 \wedge n=0 \;\rightarrow\; m, n := m-1, 1 \\
 & [\,]\;\; m>0 \wedge n>0 \;\rightarrow\; c[p] := m-1 \;;\; n, p := n-1, p+1 \\
 & \mathsf{od} \\
; & \{\; A\cdot M\cdot N = n+1 \;\} \\
 & r := n+1 \\
 & \{\; r = A\cdot M\cdot N \;\}
\end{array}
$$

□

8.12 Exercises

2003.10.6 2013.4.15 correction of a few minor errors

Chapter 9

Take-and-Drop Calculus

9.0 Take and Drop

Functions hd and tl partition a nonempty listoid into its first element and what remains after removal of that first element. This is reflected by the (known) relation, for any x in $\mathcal{P}_1$:

$$x \;=\; hd{\cdot}x \triangleright tl{\cdot}x \quad ,$$

which can also be formulated as:

$$x \;=\; [\,hd{\cdot}x\,] \mathbin{+\!\!+} tl{\cdot}x \quad .$$

This suggests a rather straightforward generalization: for any natural n, a listoid with at least n elements can be partitioned into the list of its first n elements and what remains after removal of these first n elements. This generalization turns out to have nice calculational properties; therefore, we introduce a notation for it.

Binary operator $\lceil$ – pronounced as "take" or "up to" – has the informal meaning that $x\lceil n$ is x's finite prefix of length n, whereas binary operator $\lfloor$ – "drop" or "from" – has the informal meaning that $x\lfloor n$ is what remains of x after removal of its finite prefix of length n, for all natural n. The expressions $x\lceil n$ and $x\lfloor n$ are well-defined if (and only if) x has at least n elements, that is, if $x{\in}\mathcal{P}_n$. Then, $x = x\lceil n \mathbin{+\!\!+} x\lfloor n$, where $x\lceil n$ is a finite list of length n and $x\lfloor n$ has at least m elements whenever x has at least $n{+}m$ elements. In addition, $x\lfloor n$ is a list if and only if x is a list with at least n elements.

Function $(\lceil n)$ is well-defined on domain $\mathcal{P}_n$ and, therefore, also on the domains $\mathcal{P}_{n+m}$, $\mathcal{L}_n$, $\mathcal{L}_{n+m}$, and $\mathcal{L}_\infty$, because each of the latter is included in

$\mathcal{P}_n$. Function $(\lfloor n)$ is well-defined on domain $\mathcal{P}_n$ and, for the same reasons, also on the domains $\mathcal{P}_{n+m}$, $\mathcal{L}_n$, $\mathcal{L}_{n+m}$, and $\mathcal{L}_\infty$.

First, we summarize the type relations for $\lceil$ and $\lfloor$; next, we give their most characteristic property and, finally, we give a recursive definition.

property 0 : Functions $\lceil$ and $\lfloor$ have the following types, for natural m, n ; we also give informal readings of these types:

$(\lceil n) \in \mathcal{P}_n \to \mathcal{L}_n$	"list of a listoid"
$(\lceil n) \in \mathcal{P}_{n+m} \to \mathcal{L}_n$	"first n of a listoid"
$(\lceil n) \in \mathcal{L}_n \to \mathcal{L}_n$	"identity function on $\mathcal{L}_n$"
$(\lceil n) \in \mathcal{L}_{n+m} \to \mathcal{L}_n$	"first n of a finite list"
$(\lceil n) \in \mathcal{L}_\infty \to \mathcal{L}_n$	"first n of an infinite list"
$(\lfloor n) \in \mathcal{P}_n \to \mathcal{P}_0$	"delete all elements from a listoid"
$(\lfloor n) \in \mathcal{P}_{n+m} \to \mathcal{P}_m$	"shorten a listoid by n"
$(\lfloor n) \in \mathcal{L}_n \to \mathcal{L}_0$	"delete all elements from a list"
$(\lfloor n) \in \mathcal{L}_{n+m} \to \mathcal{L}_m$	"shorten a finite list by n"
$(\lfloor n) \in \mathcal{L}_\infty \to \mathcal{L}_\infty$	"shorten an infinite list by n"

□

For example, because of its type $\mathcal{P}_n \to \mathcal{P}_0$ and because $\mathcal{P}_0$ is the whole universe of values, we cannot conclude anything about $x \lfloor n$ if $x \in \mathcal{P}_n$ is all we know about x . Similarly, because of its type $\mathcal{L}_n \to \mathcal{L}_0$ and because $\mathcal{L}_0$ only contains $[\,]$, we have $x \lfloor n = [\,]$, for all $x \in \mathcal{L}_n$.

property 1 : For natural n and $x \in \mathcal{P}_n$:

$$x \;=\; x \lceil n \mathbin{+\!\!+} x \lfloor n \quad .$$

□

definition 2 : for natural n and $x \in \mathcal{P}_n$:

$$\begin{array}{lcl} x \lceil 0 & = & [\,] \\ x \lfloor 0 & = & x \\ (b \triangleright x) \lceil (n{+}1) & = & b \triangleright x \lceil n \\ (b \triangleright x) \lfloor (n{+}1) & = & x \lfloor n \end{array}$$

□

Properties 0 and 1 can be proved from this definition. To illustrate this we prove property 1 here.

proof of property 1: By induction on n. For $x \in \mathcal{P}_0$, we derive:

$$
\begin{array}{ll}
 & x \lceil 0 \mathbin{+\!\!+} x \lfloor 0 \\
= & \quad \{ \ x \in \mathcal{P}_0 : \text{ definition 2 } \} \\
 & [\,] \mathbin{+\!\!+} x \\
= & \quad \{ \text{ property of } \mathbin{+\!\!+} \ \} \\
 & x \quad .
\end{array}
$$

For any natural n, every element of $\mathcal{P}_{n+1}$ has shape $b \triangleright x$, for some b and $x \in \mathcal{P}_n$; using this we derive:

$$
\begin{array}{ll}
 & (b \triangleright x) \lceil (n{+}1) \mathbin{+\!\!+} (b \triangleright x) \lfloor (n{+}1) \\
= & \quad \{ \ x \in \mathcal{P}_n : \text{ definition 2 } \} \\
 & (b \triangleright (x \lceil n)) \mathbin{+\!\!+} x \lfloor n \\
= & \quad \{ \text{ property of } \mathbin{+\!\!+} \ \} \\
 & b \triangleright (x \lceil n \mathbin{+\!\!+} x \lfloor n) \\
= & \quad \{ \ x \in \mathcal{P}_n : \text{ Induction Hypothesis } \} \\
 & b \triangleright x \quad .
\end{array}
$$

□

If so desired, definition 2 can be extended safely with two additional clauses, as a result of which $(\lceil n)$ and $(\lfloor n)$ are well-defined for all finite lists:

$$[\,] \lceil n \ = \ [\,] \quad \text{and:} \quad [\,] \lfloor n \ = \ [\,] \quad ,$$

but it remains to be seen whether this is of any practical use.

The take and drop operators have other useful properties, which can be proved from definition 2 as well. These proofs are left to the exercises. The first pair of properties relate the individual elements of the list(oid)s involved.

properties 3: for natural m, n, i and $x \in \mathcal{P}_{n+m}$:

$$
\begin{array}{lcll}
(x \lceil n) \cdot i & = & x \cdot i & , \text{ for } i < n \\
(x \lfloor n) \cdot i & = & x \cdot (n{+}i) & , \text{ for } i < m
\end{array}
$$

□

As a corollary, for instance, we have $x \cdot n = (x \lfloor n) \cdot 0$.

The second set contains properties about compositions of $\lceil$ and $\lfloor$. Notice that, because of the types of the operands, expressions of the shape $x \lfloor n \lceil m$ can only be read as $(x \lfloor n) \lceil m$: the other reading, $x \lfloor (n \lceil m)$, is meaningless. These rules provide us with a true take-and-drop calculus.

properties 4: for natural m, n and $x \in \mathcal{P}_{n+m}$:

$$\begin{array}{lcl} x \lceil (n+m) & = & x \lceil n \mathbin{+\!\!+} x \lfloor n \lceil m \\ x \lfloor (n+m) & = & x \lfloor n \lfloor m \\ x \lceil (n+m) \lceil n & = & x \lceil n \\ x \lceil (n+m) \lfloor n & = & x \lfloor n \lceil m \end{array}$$

□

These properties can be proved by induction on n. A proof by induction on m is likely to fail, because, for instance, in $x \lfloor n \lceil m$ the definition of $\lceil m$ cannot be applied as long we know nothing about $x \lfloor n$; on the other hand, the definition of $\lfloor n$ can be applied, regardless of the (surrounding) $\lceil m$.

9.1 Segments

A *segment* of a listoid x is any finite list "occurring somewhere in" x. This means that a segment of x is any finite list t satisfying, for some finite list s and some listoid y:

$$s \mathbin{+\!\!+} t \mathbin{+\!\!+} y = x \quad .$$

This is an implicit definition. An explicit definition of segments is possible by means of the take and drop operators. For this purpose we need the additional notions of initial segment and tail segment.

An *initial segment* of a listoid x is any finite list that is a prefix of x; so, an initial segment of $x \in \mathcal{P}_n$ is any finite list $x \lceil j$, for $j \leq n$. A *tail segment* of a finite list x is any finite list that remains after removal of some prefix of x; hence, a tail segment of $x \in \mathcal{L}_n$ is any finite list $x \lfloor i$, for $i \leq n$. Notice that if x is only a listoid (and not a finite list) then $x \lfloor i$ is only a listoid too (and not a list); in this case we simply call $x \lfloor i$ a *tail* of x.

Segments can now be defined in two ways: a segment of x either is a tail segment of an initial segment of x, or it is an initial segment of a tail of x. In formula: a segment of $x \in \mathcal{P}_n$ either is any finite list $x \lceil (j+i) \lfloor i$ or is any

finite list $x\lfloor i\lceil j$, in either case for $0 \leq i \leq i+j \leq n$. Both formulae define the same segment, namely the segment *of length* j occurring in x *at position* i.

It is awkward that segments must be defined in either of two ways, each of which destroys the symmetry. In this respect, the above implicit definition is more neutral, thanks to the associativity of concatenation. Fortunately, however, the situation is not that awkward: by the last of properties 4 we have $x\lceil(j+i)\lfloor i = x\lfloor i\lceil j$; hence, we can always choose the simpler one, in the safe knowledge that we can simply switch to the other form, if necessary.

The segment $x\lfloor i\lceil j$ consists of the elements $x \cdot h$, for all $h : i \leq h < i+j$.

9.2 Example: the maximal segment sum

For the sake of comparison we present two specifications for the well-known problem of the maximal segment sum, without and with the use of the take and drop operators. From these specifications we will derive solutions.

9.2.0 an implicit specification

A segment of a finite list s is any finite list y satisfying, for some finite lists x and z: $x +\!\!+ y +\!\!+ z = s$. Thus we obtain a concise specification for the problem of the maximal segment sum, where function *sum* maps a finite integer list to the sum of its elements.

specification 4: for $s \in \mathcal{L}_*(\mathsf{Int})$:

$$f \cdot s \;=\; (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = s : sum \cdot y\,) \quad .$$

□

Using this specification we derive:

$$\begin{array}{ll}
 & f \cdot [\,] \\
= & \quad \{ \text{ specification 4 } \} \\
 & (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = [\,] : sum \cdot y\,) \\
= & \quad \{ \text{ property of } +\!\!+ \ \} \\
 & (\,\mathsf{max}\, x, y, z : x = [\,] \wedge y = [\,] \wedge z = [\,] : sum \cdot y\,) \\
= & \quad \{ \text{ one-point rule } \} \\
 & sum \cdot [\,] \\
= & \quad \{ \text{ property of } sum \ \}
\end{array}$$

$$0 \quad ,$$

and:

$$
\begin{array}{ll}
 & f\cdot(b \triangleright s) \\
= & \quad \{ \text{ specification 4 } \} \\
 & (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = b \triangleright s : sum\cdot y\,) \\
= & \quad \{ \text{ range split: } x = [\,] \text{ or } x := c \triangleright x \ \} \\
 & (\,\mathsf{max}\, y, z : y +\!\!+ z = b \triangleright s : sum\cdot y\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, c, x, y, z : (c \triangleright x) +\!\!+ y +\!\!+ z = b \triangleright s : sum\cdot y\,) \\
= & \quad \{ \bullet \text{ new function } g \text{ ; property of } +\!\!+ \ \} \\
 & g\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\, c, x, y, z : c \triangleright (x +\!\!+ y +\!\!+ z) = b \triangleright s : sum\cdot y\,) \\
= & \quad \{ \text{ property of } \triangleright \ \} \\
 & g\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\, c, x, y, z : c = b \wedge x +\!\!+ y +\!\!+ z = s : sum\cdot y\,) \\
= & \quad \{ \text{ one-point rule } \} \\
 & g\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = s : sum\cdot y\,) \\
= & \quad \{ \text{ specification 4, by Ind. Hyp. } \} \\
 & g\cdot(b \triangleright s) \ \mathsf{max} \ f\cdot s \quad .
\end{array}
$$

Here the new function g has the following specification; it maps an integer list to the maximal sum of all its initial segments.

specification 5: for $s \in \mathcal{L}_*(\mathsf{Int})$:

$$g\cdot s \ = \ (\,\mathsf{max}\, y, z : y +\!\!+ z = s : sum\cdot y\,) \quad .$$

□

In a way very similar to what we did for $f\cdot[\,]$ we can derive that $g\cdot[\,] = 0$ satisfies the specification; furthermore:

$$
\begin{array}{ll}
 & g\cdot(b \triangleright s) \\
= & \quad \{ \text{ specification 5 } \} \\
 & (\,\mathsf{max}\, y, z : y +\!\!+ z = b \triangleright s : sum\cdot y\,) \\
= & \quad \{ \text{ range split: } y = [\,] \text{ or } y := c \triangleright y \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\,\mathsf{max}\, z : z = b \triangleright s : sum \cdot [\,]\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, c, y, z : (c \triangleright y) +\!\!+ z = b \triangleright s : sum \cdot (c \triangleright y)\,) \\
= & \quad \{ \text{ one-point rule; property of } +\!\!+ \ \} \\
 & sum \cdot [\,] \quad \mathsf{max} \quad (\,\mathsf{max}\, c, y, z : c \triangleright (y +\!\!+ z) = b \triangleright s : sum \cdot (c \triangleright y)\,) \\
= & \quad \{ \text{ property of } \triangleright \text{ ; one-point rule } \} \\
 & sum \cdot [\,] \quad \mathsf{max} \quad (\,\mathsf{max}\, y, z : y +\!\!+ z = s : sum \cdot (b \triangleright y)\,) \\
= & \quad \{ \text{ properties of } sum \ \} \\
 & 0 \quad \mathsf{max} \quad (\,\mathsf{max}\, y, z : y +\!\!+ z = s : \ b + sum \cdot y\,) \\
= & \quad \{ \ (b+) \text{ over } \mathsf{max} \ \} \\
 & 0 \ \mathsf{max} \ (b + (\,\mathsf{max}\, y, z : y +\!\!+ z = s : sum \cdot y)) \\
= & \quad \{ \text{ specification 5, by Ind. Hyp. } \} \\
 & 0 \ \mathsf{max} \ (b + g \cdot s) \quad .
\end{array}
$$

Thus we obtain the following recursive declarations for f and g. This solution has quadratic time complexity, but by means of the standard tupling technique the declarations for f and g be combined into a single one, with linear time complexity.

declaration 6:

$$
\begin{array}{lll}
 & f \cdot [\,] & = \ 0 \\
\& & f \cdot (b \triangleright s) & = \ g \cdot (b \triangleright s) \ \mathsf{max} \ f \cdot s \\
\& & g \cdot [\,] & = \ 0 \\
\& & g \cdot (b \triangleright s) & = \ 0 \ \mathsf{max} \ (b + g \cdot s)
\end{array}
$$

□

9.2.1 a specification with take and drop

For s a finite list and for natural i, j, with $i+j \leq \# s$, the finite list $s \lfloor i \lceil j$ is the segment of s of length j and at position i. Thus we can specify the problem of the maximal segment sum as follows.

specification 7: for $s \in \mathcal{L}_n(\mathsf{Int})$ (and natural n):

$$
f \cdot s \ = \ (\,\mathsf{max}\, i, j : 0 \leq i \leq i+j \leq n : sum \cdot (s \lfloor i \lceil j)\,) \quad .
$$

□

Using this specification we derive:

$$
\begin{array}{ll}
 & f\cdot[\,] \\
= & \quad \{ \text{ specification 7, with } n=0 \ \} \\
 & (\,\mathsf{max}\, i,j : 0\le i\le i+j\le 0 : sum\cdot([\,]\lfloor i\lceil j)\,) \\
= & \quad \{ \text{ algebra } \} \\
 & (\,\mathsf{max}\, i,j : i=0 \wedge j=0 : sum\cdot([\,]\lfloor i\lceil j)\,) \\
= & \quad \{ \text{ one-point rule } \} \\
 & sum\cdot([\,]\lfloor 0\lceil 0) \\
= & \quad \{ \text{ definition of } \lceil \ \} \\
 & sum\cdot[\,] \\
= & \quad \{ \text{ property of } sum \ \} \\
 & 0 \quad ,
\end{array}
$$

and for $b\triangleright s\in\mathcal{L}_{n+1}(\mathsf{Int})$:

$$
\begin{array}{ll}
 & f\cdot(b\triangleright s) \\
= & \quad \{ \text{ specification 4 } \} \\
 & (\,\mathsf{max}\, i,j : 0\le i\le i+j\le n+1 : sum\cdot((b\triangleright s)\lfloor i\lceil j)\,) \\
= & \quad \{ \text{ range split: } i=0 \text{ or } i:=i+1 \ \} \\
 & (\,\mathsf{max}\, j : 0\le j\le n+1 : sum\cdot((b\triangleright s)\lfloor 0\lceil j)\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, i,j : 1\le i+1\le i+1+j\le n+1 : sum\cdot((b\triangleright s)\lfloor (i+1)\lceil j)\,) \\
= & \quad \{ \text{ definition } \lfloor \text{ , twice; simplification } \} \\
 & (\,\mathsf{max}\, j : 0\le j\le n+1 : sum\cdot((b\triangleright s)\lceil j)\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, i,j : 0\le i\le i+j\le n : sum\cdot(s\lfloor i\lceil j)\,) \\
= & \quad \{ \bullet \text{ new function } g \text{ ; specification 4, by Ind. Hyp. } \} \\
 & g\cdot(b\triangleright s) \ \mathsf{max} \ f\cdot s \quad .
\end{array}
$$

Here the new function g has the following specification.

specification 8: for $s\in\mathcal{L}_n(\mathsf{Int})$ (and natural n):

$$g\cdot s \ = \ (\,\mathsf{max}\, j : 0\le j\le n : sum\cdot(s\lceil j)\,) \quad .$$

□

In a way very similar to what we did for $f\cdot[\,]$ we can derive that $g\cdot[\,]=0$ satisfies the specification; furthermore:

$$
\begin{array}{ll}
 & g\cdot(b\triangleright s) \\
= & \quad \{ \text{ specification } 8 \ \} \\
 & (\,\mathsf{max}\, j : 0\le j\le n{+}1 : sum\cdot((b\triangleright s)\lceil j)\,) \\
= & \quad \{ \text{ range split: } j=0 \text{ or } j:=j+1 \ \} \\
 & sum\cdot((b\triangleright s)\lceil 0) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, j : 1\le j{+}1\le n{+}1 : sum\cdot((b\triangleright s)\lceil(j{+}1))\,) \\
= & \quad \{ \text{ definition } \lceil \text{ , twice; simplification } \} \\
 & sum\cdot[\,] \quad \mathsf{max} \quad (\,\mathsf{max}\, j : 0\le j\le n : sum\cdot(b\triangleright(s\lceil j))\,) \\
= & \quad \{ \text{ properties of } sum \ \} \\
 & 0 \quad \mathsf{max} \quad (\,\mathsf{max}\, j : 0\le j\le n : b+sum\cdot(s\lceil j)\,) \\
= & \quad \{ \ (b+) \text{ over } \mathsf{max} \ \} \\
 & 0 \ \mathsf{max} \ (b+(\,\mathsf{max}\, j : 0\le j\le n : sum\cdot(s\lceil j)\,)) \\
= & \quad \{ \text{ specification } 8, \text{ by Ind. Hyp. } \} \\
 & 0 \ \mathsf{max} \ (b+g\cdot s) \quad .
\end{array}
$$

Thus we obtain exactly the same recursive declarations for f and g as in the previous case: except for the differences in the notation, the two solutions are identical; see declaration 6.

9.3 Exercises

1. Prove that, for all natural n, function $(\lceil n)$ has type $\mathcal{L}_\infty\to\mathcal{L}_n$ and that $(\lfloor n)$ has type $\mathcal{L}_\infty\to\mathcal{L}_\infty$; use the definitions of $\mathcal{L}_n$ and $\mathcal{L}_\infty$.

2. Prove properties 3 and 4.

3. Prove, for natural i,n, for $x\in\mathcal{L}_n$, and for any y:

$$
\begin{array}{lcll}
(x{+\!\!+}y)\lceil i & = & x\lceil i & \text{, if } i\le n \\
(x{+\!\!+}y)\lceil i & = & x{+\!\!+}y\lceil(i{-}n) & \text{, if } n<i \\
 & & & \\
(x{+\!\!+}y)\lfloor i & = & x\lfloor i{+\!\!+}y & \text{, if } i\le n \\
(x{+\!\!+}y)\lfloor i & = & y\lfloor(i{-}n) & \text{, if } n<i
\end{array}
$$

4. Prove, for $x\in\mathcal{P}_{n+1}$:

$$
rev\cdot(x\lceil(n{+}1)) \ = \ x\cdot n \triangleright rev\cdot(x\lceil n)
$$

5. Formulate and prove relations connecting $\lceil$ and $\lfloor$ with $\lhd$.

2002.4.11 2002.4.23

Chapter 10

Infinite Lists

Only two things are infinite, the universe and human stupidity, and I am not sure about the former.
Albert Einstein

10.0 Introduction

In this chapter we develop some theory to facilitate reasoning about (programs with) infinite lists. Proving properties of infinite lists always involves mathematical induction, but the patterns of this induction can be captured in a few useful theorems. This is very similar to the Invariance Theorem in sequential programming, which captures the induction needed for reasoning about repetitions.

Because an infinite list has no (finite) length, proofs by induction on the length of lists are impossible. Every infinite list with element type B, however, also is a function of type $\mathsf{Nat} \to B$, and this allows proofs by induction on Nat, which we view as the list's domain. Properties of infinite lists can be proved either by straightforward mathematical induction on this domain or by means of the theorems from this chapter. Which is more convenient depends on the particular problem and its setting, but both approaches are essentially the same. In *derivations* straightforward induction will be prevailing whereas the theorems are applicable for *a posteriori* proofs. We will see examples of both.

* * *

From Chapter 6 we recall the definition of the sets $\mathcal{P}_n(B)$ – the listoids of order n – and $\mathcal{L}_\infty(B)$ – the infinite lists –, where B is the type of the elements.

definition 0 : For all t in Ω and for all natural n we have:

$$\begin{array}{lcl} t\in\mathcal{P}_0(B) & \equiv & \mathsf{true} \\ t\in\mathcal{P}_{n+1}(B) & \equiv & (\exists b,s : b\in B \wedge s\in\mathcal{P}_n(B) : t = b\triangleright s) \\ t\in\mathcal{L}_\infty(B) & \equiv & (\forall n :: t\in\mathcal{P}_n(B)) \end{array}$$

□

All elements of $\mathcal{P}_{n+1}(B)$ have the shape $b\triangleright s$; often, we use this implicitly by calculating with this shape right away, and we use properties like:

$$(b\triangleright s)\in\mathcal{P}_{n+1}(B) \;\equiv\; b\in B \wedge s\in\mathcal{P}_n(B) \quad .$$

As before, we often omit the element type B from our formulae, either because it is clear from the context or because it is irrelevant.

10.1 Equality of Infinite Lists

Finite lists (of equal length) have the property that they are equal if all of their elements are equal. For infinite lists we cannot *prove* a similar property, but nevertheless we wish to consider infinite lists equal whenever all of their elements are equal. As the formalism provides no ways to distuinguish such lists, we can safely *postulate* this to be so, that is, we just *define* equality of infinite lists as equality of all of their elements.

definition 1 (Infinite List Equality) : For infinite lists x, y:

$$x = y \;\equiv\; (\forall i :: x\cdot i = y\cdot i) \quad .$$

□

Occasionally, it is more convenient to rephrase this definition in terms of equality of all finite prefixes. This yields an equivalent formulation of the same definition. We will use both versions of this definition just as we see fit.

definition 2 (Infinite List Equality) : For infinite lists x, y:

$$x = y \;\equiv\; (\forall j :: x\lceil j = y\lceil j) \quad .$$

□

10.2 Productivity

Declarations of infinite lists always are recursive. For example, we may define an infinite list x all whose elements are equal to 7 in the following, recursive way – in Section 6.1.3 we have already derived this –:

$$x \;=\; 7 \triangleright x \quad .$$

Questions to be answered now are: why would a definition of this shape yield an infinite list and why would that list have the required properties? As to these questions, by repeated unfolding of x we obtain:

$$\begin{array}{ll}
 & x \\
= & \quad \{ \text{ definition of } x \text{ } \} \\
 & 7 \triangleright x \\
= & \quad \{ \text{ definition of } x \text{ } \} \\
 & 7 \triangleright (7 \triangleright x) \\
= & \quad \{ \text{ definition of } x \text{ } \} \\
 & 7 \triangleright (7 \triangleright (7 \triangleright x)) \\
= & \quad \{ \text{ definition of } x \text{ } \} \\
 & 7 \triangleright (7 \triangleright (7 \triangleright (7 \triangleright x))) \quad , \\
 & \text{and so on.}
\end{array}$$

This calculation can be continued indefinitely: we get the impression that this definition yields an infinite list indeed, and that all its elements are 7.

On the other hand, consider this recursive definition, where K_7 denotes a function with constant value 7 – that is: $K_7 \cdot b = 7$, for all b –:

$$y \;=\; K_7 \bullet y \quad .$$

If this y would be an infinite list – it isn't –, we could derive, for natural i:

$$\begin{array}{ll}
 & y \cdot i \\
= & \quad \{ \text{ definition of } y \text{ } \} \\
 & (K_7 \bullet y) \cdot i \\
= & \quad \{ \text{ property of } \bullet \text{, (only) if } y \text{ is an infinite list } \}
\end{array}$$

$$
\begin{array}{ll}
 & K_7 \cdot (y \cdot i) \\
= & \quad \{ \text{ definition of } K_7 \ \} \\
 & 7 \quad ,
\end{array}
$$

from which we conclude that here, too, all y's elements are 7. There is no way, however, to conclude that y should be an infinite list, and rightly so: viewed as an equation, this definition also admits any finite list containing 7's as a solution! In addition, this equation is likely to have other, "undefined" solutions as well, because viewed as a computation rule y's definition will only yield expressions like:

$$
\begin{array}{ll}
 & y \\
= & \quad \{ \text{ definition of } y \ \} \\
 & K_7 \bullet y \\
= & \quad \{ \text{ definition of } y \ \} \\
 & K_7 \bullet (K_7 \bullet y) \\
= & \quad \{ \text{ definition of } y \ \} \\
 & K_7 \bullet (K_7 \bullet (K_7 \bullet y)) \\
= & \quad \{ \text{ definition of } y \ \} \\
 & K_7 \bullet (K_7 \bullet (K_7 \bullet (K_7 \bullet y))) \quad , \\
 & \text{and so on indefinitely.}
\end{array}
$$

These expressions never yield anything useful because the computation rule for $\bullet$ only becomes applicable after it has been established that its second operand either is $[\,]$ or has the shape $b \triangleright s$, and neither shape will ever emerge.

In this example, every unfolding of x's definition yields a new element and an occurrence of the operator $\triangleright$, whereas every unfolding of y's definition only yields a more complicated expression of the shape $K_7 \bullet \cdots$, which leads nowhere. To distinguish them we call definitions like the one for x *productive*: productive definitions yield infinite lists.

* * *

We now define *productivity* formally. Productivity entails two properties, labelled (α) and (β) below; property (α) guarantees that the fixed points of a function are infinite lists, whereas (β) guarantees uniqueness of these fixed points; this is useful for proving properties of the infinite lists thus defined.

The informal idea is that a function is productive if its value has at least one element more than its argument.

definition 3 (Productivity) : A function F, of type $\Omega \to \Omega$, is *productive* if it has the following two properties:

(α) $(\forall x, j :: x \in \mathcal{P}_j \Rightarrow F \cdot x \in \mathcal{P}_{j+1})$

(β) $(\forall x, y, j : x, y \in \mathcal{L}_\infty : x \lceil j = y \lceil j \Rightarrow F \cdot x \lceil (j+1) = F \cdot y \lceil (j+1))$

□

The following example gives the prototypical way for constructing productive functions: by means of the operator $\triangleright$.

example: For any value b, function F is productive, if defined by:

$$F \cdot x \;=\; b \triangleright x \quad .$$

□

property 4 : If function F, of type $\Omega \to \Omega$, is productive then F has type $\mathcal{L}_\infty \to \mathcal{L}_\infty$.

□

Productivity is important because of the following theorem. It states that all fixed points of a productive function are infinite lists, and that all these lists are equal; in other words, a productive function has a *unique* infinite list as its fixed point.

theorem 5 (First Productivity Theorem) : Productive functions F, of type $\Omega \to \Omega$, satisfy:

(α) $(\forall x : x = F \cdot x : x \in \mathcal{L}_\infty)$

(β) $(\forall x, y : x = F \cdot x \wedge y = F \cdot y : x = y)$

proof: To prove part (α) we derive for any x satisfying $x = F \cdot x$:

$$\begin{array}{ll} & x \in \mathcal{L}_\infty \\ \equiv & \quad \{ \text{ definition of } \mathcal{L}_\infty \ \} \end{array}$$

$$
\begin{array}{ll}
 & (\forall j :: x \in \mathcal{P}_j) \\
\equiv & \quad \{ \text{ mathematical induction } \} \\
 & x \in \mathcal{P}_0 \ \wedge \ (\forall j :: x \in \mathcal{P}_j \Rightarrow x \in \mathcal{P}_{j+1}) \\
\equiv & \quad \{ \ x = F \cdot x \ \} \\
 & x \in \mathcal{P}_0 \ \wedge \ (\forall j :: x \in \mathcal{P}_j \Rightarrow F \cdot x \in \mathcal{P}_{j+1}) \\
\equiv & \quad \{ \text{ definition } \mathcal{P}_0 \, ; \, F \text{ is productive, part } (\alpha) \ \} \\
 & \mathsf{true} \ .
\end{array}
$$

To prove part (β) we observe that, for x, y with $x = F \cdot x \wedge y = F \cdot y$, we have, on account of (α), that $x \in \mathcal{L}_\infty$ and $y \in \mathcal{L}_\infty$. Hence, by definition 2 we have that $x = y$ is equivalent to $(\forall j :: x \lceil j = y \lceil j)$. This, in turn, is proved by mathematical induction, very much in the same way as we proved $x \in \mathcal{L}_\infty$, but now with use of part (β) of F's productivity (and the definition of $(\lceil 0)$).

□

example: Function F, defined by $F \cdot x = 7 \triangleright x$, is productive and, hence, value s, defined by $s = 7 \triangleright s$, is an infinite list. That such s satisfies $(\forall i :: s \cdot i = 7)$ can be proved separately, either directly by mathematical induction or by the following Second Productivity Theorem.

□

10.3 Admissible Predicates

To prove properties of infinite lists – like $(\forall i :: s \cdot i = 7)$ in the last example – in a smooth way, we also would like to have a theorem that streamlines the induction argument. Unfortunately, we can only formulate such a theorem for a restricted class of properties, the so-called *admissible predicates*; fortunately, however, many useful properties are admissible predicates indeed.

definition 6 (Admissibility) : A predicate P on $\mathcal{L}_\infty$ – so P has type $\mathcal{L}_\infty \rightarrow \mathsf{Bool}$ – is *admissible* if a sequence $Q_j (0 \leq j)$ of predicates on $\mathcal{L}_\infty$ exists with the following two properties:

(α) $\quad (\forall x : x \in \mathcal{L}_\infty : \ P \cdot x \ \equiv \ (\forall j :: Q_j \cdot x))$

(β) $\quad (\forall x, y, j : x, y \in \mathcal{L}_\infty : \ x \lceil j = y \lceil j \ \Rightarrow \ (Q_j \cdot x \equiv Q_j \cdot y))$

(In words: P is the conjunction of an infinite sequence of *approximations* $Q_j(0 \leq j)$, where formula (β) expresses that, effectively, $Q_j \cdot x$ depends on $x \lceil j$ only.

□

property 7: If *P0* and *P1* are admissible then so is $P0 \wedge P1$.

□

examples: The predicate $(\forall i :: x \cdot i = 7)$ is admissible: just let $Q_j \cdot x$ be $(\forall i : i < j : x \cdot i = 7)$. The predicate (in x and for fixed y) and $x = y$ is admissible: take $(\forall i : i < j : x \cdot i = y \cdot i)$ for $Q_j \cdot x$. Consequently, "being unique" is admissible too. The negations of these predicates, that is $(\exists i :: x \cdot i \neq 7)$ and $x \neq y$, are *not* admissible. Both the constant predicates true and false are admissible.

Also, for any type B, the predicate "having type $\mathcal{L}_\infty(B)$" is the conjunction of "having type $\mathcal{L}_\infty$" and "having type $\mathsf{Nat} \to B$", because, for any x we have:

$$x \in \mathcal{L}_\infty(B) \;\equiv\; x \in \mathcal{L}_\infty \;\wedge\; x \in \mathsf{Nat} \to B \quad .$$

The conjunct $x \in \mathsf{Nat} \to B$ is admissible; hence, once we have established $x \in \mathcal{L}_\infty$, we can prove separately that all of x's elements have type B and, if so desired, we can do so by an appeal to the theorem.

□

The theorem now states conditions under which the fixed points of a productive function satisfy an admissible predicate. These conditions resemble the conditions for a proof by mathematical induction; the proof pattern embodied in this theorem is also known as *Fixed-Point Induction*.

theorem 8 (Second Productivity Theorem): For productive function F and admissible predicate P:

$$\begin{array}{ll} \text{if} & (\exists x : x \in \mathcal{L}_\infty : P \cdot x) \;\text{ and }\; (\forall x : x \in \mathcal{L}_\infty : P \cdot x \Rightarrow P \cdot (F \cdot x)) \\ \text{then} & (\forall x : x = F \cdot x : P \cdot x) \end{array}$$

proof: With $Q_j(0 \leq j)$ as in Definition 6, we introduce an infinite sequence $s_j(0 \leq j)$ of infinite lists, thus:

$$P\cdot s_0 \;\wedge\; (\forall j :: s_{j+1} = F\cdot s_j) \quad .$$

List s_0 exists on account of the premiss $(\exists x : x\in\mathcal{L}_\infty : P\cdot x)$. Now let x satisfy $x = F\cdot x$. Then, by F's productivity we have $x\in\mathcal{L}_\infty$ and now it is a matter of straightforward mathematical induction to establish the following properties of s – this also involves the second premiss and F's productivity –:

(0) $(\forall j :: s_j \in \mathcal{L}_\infty)$

(1) $(\forall j :: P\cdot s_j)$

(2) $(\forall j :: x\lceil j = s_j\lceil j)$

Now we are ready for the main proof:

$$\begin{array}{ll}
 & P\cdot x \\
\equiv & \quad \{\ P \text{ is admissible } (\alpha)\ \} \\
 & (\forall j :: Q_j\cdot x) \\
\equiv & \quad \{\ P \text{ is admissible } (\beta)\text{, using (0) and (2) }\} \\
 & (\forall j :: Q_j\cdot s_j) \\
\Leftarrow & \quad \{\ \text{diagonalization }\} \\
 & (\forall i, j :: Q_j\cdot s_i) \\
\equiv & \quad \{\ P \text{ is admissible } (\alpha)\text{, using (0) }\} \\
 & (\forall i :: P\cdot s_i) \\
\equiv & \quad \{\ (1)\ \} \\
 & \mathsf{true} \quad .
\end{array}$$

□

example: With F defined by $F\cdot x = 7\triangleright x$, both the predicate (in x) $(\forall i :: x\cdot i = 7)$ and its negation $(\exists i :: x\cdot i \neq 7)$ satisfy the requirements of the theorem. The former is admissible whereas the latter is not. Indeed, the infinite list s, defined by $s = 7\triangleright s$, satisfies $(\forall i :: s\cdot i = 7)$ whereas it does not satisfy $(\exists i :: s\cdot i \neq 7)$. This shows that we cannot do without the requirement of admissibility.

□

10.4 Uniform Productivity

The notion of productivity developed in the previous section caters for simple recursive definitions of (constant) infinite lists. A more interesting notion of productivity pertains to functions with an additional parameter, of some type V. Such functions enable us to construct recursive declarations of functions of type $V \to \mathcal{L}_\infty$: functions that have infinite lists as their values. If the recursion pattern and, hence, the productivity do not depend on the value of the parameter we speak of *uniform productivity*.

For some given types V and B, we are heading for (recursive) declarations of functions of type $V \to \mathcal{L}_\infty(B)$. As no further restrictions are imposed, V itself may be a set of infinite lists. Hence, the patterns described here also cover functions from infinite lists to infinite lists.

For some given functions h, of type $V \to B$, and k, of type $V \to V$, the prototype of a uniformly productive function is a function F, of type $\Omega \to V \to \Omega$, defined as follows:

$$(0) \qquad F \cdot f \cdot v \;=\; h \cdot v \triangleright f \cdot (k \cdot v) \quad .$$

Now the First Productivity Theorem can be generalized, as follows.

theorem 9 (First Uniform Productivity Theorem): Uniformly productive functions F, as defined by (0), satisfy:

$$(\alpha) \qquad (\forall f : f = F \cdot f : (\forall v : v \in V : f \cdot v \in \mathcal{L}_\infty))$$

$$(\beta) \qquad (\forall f, g : f = F \cdot f \wedge g = F \cdot g : (\forall v : v \in V : f \cdot v = g \cdot v))$$

proof: To prove part (α) we derive, for f satisfying $f = F \cdot f$ and where dummy v has type V:

$$\begin{array}{ll}
 & (\forall v :: f \cdot v \in \mathcal{L}_\infty) \\
\equiv & \quad \{ \text{ definition of } \mathcal{L}_\infty \ \} \\
 & (\forall v :: (\forall j :: f \cdot v \in \mathcal{P}_j)) \\
\equiv & \quad \{ \text{ exchanging dummies } \} \\
 & (\forall j :: (\forall v :: f \cdot v \in \mathcal{P}_j)) \\
\equiv & \quad \{ \text{ mathematical induction over } j \ \} \\
 & (\forall v :: f \cdot v \in \mathcal{P}_0) \ \wedge \ (\forall j :: (\forall v :: f \cdot v \in \mathcal{P}_j) \Rightarrow (\forall v :: f \cdot v \in \mathcal{P}_{j+1})) \\
\equiv & \quad \{ \text{ definition } \mathcal{P}_0 \ \}
\end{array}$$

$$
\begin{array}{ll}
 & (\forall j :: (\forall v :: f\cdot v \in \mathcal{P}_j) \Rightarrow (\forall v :: f\cdot v \in \mathcal{P}_{j+1})) \\
\equiv & \quad \{ f = F\cdot f \} \\
 & (\forall j :: (\forall v :: f\cdot v \in \mathcal{P}_j) \Rightarrow (\forall v :: F\cdot f\cdot v \in \mathcal{P}_{j+1})) \\
\equiv & \quad \{ \text{definition (0) (of } F\text{)} \} \\
 & (\forall j :: (\forall v :: f\cdot v \in \mathcal{P}_j) \Rightarrow (\forall v :: h\cdot v \triangleright f\cdot(k\cdot v) \in \mathcal{P}_{j+1})) \\
\equiv & \quad \{ \text{definition of } \mathcal{P}_{j+1} \} \\
 & (\forall j :: (\forall v :: f\cdot v \in \mathcal{P}_j) \Rightarrow (\forall v :: f\cdot(k\cdot v) \in \mathcal{P}_j)) \\
\equiv & \quad \{ k \in V \to V : \text{ predicate calculus} \} \\
 & \mathsf{true} \ .
\end{array}
$$

To prove part (β) we observe that, for f, g with $f = F\cdot f \wedge g = F\cdot g$, we have, on account of (α), that f and g have type $V \to \mathcal{L}_\infty$. Hence, by definition 2, $f\cdot v = g\cdot v$ is equivalent to $(\forall j :: f\cdot v \lceil j = g\cdot v \lceil j)$. This, in turn, is proved by mathematical induction, very much in the same way as we proved $f\cdot v \in \mathcal{L}_\infty$, but now with use of part (β) of F's productivity (and the definition of $(\lceil 0)$).

□

example 10: Function *from* defined by:

$$from\cdot n \ = \ n \triangleright from\cdot(n{+}1) \quad ,$$

is the fixed point of function F –so, $from = F\cdot from$– defined by:

$$F\cdot f\cdot n \ = \ n \triangleright f\cdot(n{+}1) \quad ,$$

which has the shape of definition (0), if we take the identity function for h and $(+1)$ for k. So, F is uniformly productive and, hence, *from* has type $\mathsf{Nat} \to \mathcal{L}_\infty(\mathsf{Nat})$.

□

10.5 Uniform Admissibility

Uniform productivity is a generalization of simple productivity: the simple domain $\mathcal{L}_\infty$ is generalized to $V \to \mathcal{L}_\infty$. By generalizing the notion of admissibility accordingly, we obtain the following definition of admissibility, which is useful for uniformly productive definitions.

definition 11 (Uniform Admissibility): A predicate P on $V \to \mathcal{L}_\infty$ is *admissible* if a sequence $Q_j(0 \leq j)$ of predicates on $V \to \mathcal{L}_\infty$ exists with the following two properties:

(α) $\quad (\forall f :: P \cdot f \equiv (\forall j :: Q_j \cdot f))$

(β) $\quad (\forall f, g, j :: (\forall v :: f \cdot v \lceil j = g \cdot v \lceil j) \Rightarrow (Q_j \cdot f \equiv Q_j \cdot g))$

□

property 12: If *P0* and *P1* are admissible then so is *P0* $\wedge$ *P1*.
□

lemma 13: Suppose that, for sequences $Q_j(0 \leq j)$ and $R_j(0 \leq j)$ of predicates, predicate P satisfies:

(α) $\quad (\forall f :: P \cdot f \equiv (\forall j :: Q_j \cdot f))$

(γ) $\quad (\forall j, f :: Q_j \cdot f \equiv (\forall v :: R_j \cdot (f \cdot v \lceil j) \cdot v))$

Then predicate P is admissible. To prove this it suffices to prove $(\gamma) \Rightarrow (\beta)$, which is easy – just an appeal to Leibniz –.
□

Without any further difficulties, this leads to a corresponding generalization of the Second Productivity Theorem.

theorem 14 (Second Uniform Productivity Theorem): For F a uniformly productive function, as defined by (0), and for uniformly admissible predicate P on $V \to \mathcal{L}_\infty$:

if $\quad (\exists f :: P \cdot f)$ and $(\forall f :: P \cdot f \Rightarrow P \cdot (F \cdot f))$

then $\quad (\forall f : f = F \cdot f : P \cdot f)$

proof: Entirely similar to the proof of Theorem 8.
□

example: The predicate $(\forall n, i :: f \cdot n \cdot i = n + i)$, on $\mathsf{Nat} \to \mathcal{L}_\infty(\mathsf{Nat})$, is admissible: take $(\forall n, i : i < j : f \cdot n \cdot i = n + i)$, for $Q_j \cdot f$. We now consider function *from*, as in example 10: *from* is a fixed point of func-

tion F, defined by $F\cdot f\cdot n = n \triangleright f\cdot(n{+}1)$. The predicate and this function satisfy the conditions of the theorem and, therefore, *from* satisfies:

$$(\forall n, i :: \mathit{from}\cdot n\cdot i = n{+}i) \quad .$$

In particular, $\mathit{from}\cdot 0$ satisfies $(\forall i :: \mathit{from}\cdot 0\cdot i = i)$, that is, $\mathit{from}\cdot 0$ is the infinite list of natural numbers.

□

10.6 Non-Uniform Productivity

For some type V, we assume given a function b, of type $V \to \mathsf{Bool}$, a function h, of type $V \to B$, and functions k and l, of type $V \to W$. Defined in terms of these functions, we consider a function F, as follows:

$$(1) \qquad F\cdot f\cdot v \;=\; \begin{array}{l} \mathsf{if}\ \neg b\cdot v \to f\cdot(l\cdot v) \\ \quad [\!]\ \ b\cdot v \to h\cdot v \triangleright f\cdot(k\cdot v) \\ \mathsf{fi} \end{array}$$

For $v \in V$ such that $b\cdot v$ we have that $F\cdot f\cdot v$ produces at least one element, namely $h\cdot v$, but if $\neg b\cdot v$ we cannot guarantee this. To what extent this F can be considered productive now depends on its constituents b, k, and l. We investigate what is a sufficient condition to consider a function like this nevertheless productive.

Let f be a fixed point of F. This means that $f = F\cdot f$, and that, hence, f satisfies, for all $v \in V$:

$$(2) \qquad f\cdot v \;=\; \begin{array}{l} \mathsf{if}\ \neg b\cdot v \to f\cdot(l\cdot v) \\ \quad [\!]\ \ b\cdot v \to h\cdot v \triangleright f\cdot(k\cdot v) \\ \mathsf{fi} \end{array}$$

For $v \in V$ such that $(\forall i :: \neg b\cdot(l^i\cdot v))$, all we can conclude about $f\cdot v$ is:

$$f\cdot v \;=\; f\cdot(l^i\cdot v) \quad , \text{ for all natural } i \quad .$$

This admits any value in Ω as a solution (for $f\cdot v$) and, hence, defines nothing. So, if definitions like (2) are to be of any use at all, we will have to restrict ourselves to those values v in V that do not satisfy $(\forall i :: \neg b\cdot(l^i\cdot v))$.

Therefore, we assume set V and functions b and l to be such that:

(3) $\quad (\exists i :: b\cdot(l^i\cdot v))$, for every $v\in V$.

We now prove that under this assumption, definition (1) can indeed be considered productive, and that it has a unique function of type $V\to\mathcal{L}_\infty(B)$, as its solution. We do so by transforming definition (2) into an equivalent but uniformly productive one.

Because of assumption (3) we can associate with every $v\in V$ the smallest natural number n such that $b\cdot(l^n\cdot v)$; so, n is (completely) characterized by:

$$(\forall i : i<n : \neg b\cdot(l^i\cdot v)) \ \wedge\ b\cdot(l^n\cdot v) \quad .$$

This expresses that n-fold application of function l to v yields a value satisfying b; in addition, we have $f\cdot(l^i\cdot v) = f\cdot(l^{i+1}\cdot v)$, for every $i : 0\le i<n$, because $\neg b\cdot(l^i\cdot v)$.

We now define function φ, of type $V\to V$, by $\varphi\cdot v = l^n\cdot v$, where n is the minimal value specified above. Function φ then has the following properties:

$$(\forall v :: b\cdot(\varphi\cdot v)) \ \text{ and: } \ (\forall v :: f\cdot v = f\cdot(\varphi\cdot v)) \quad .$$

Now we derive, for any $v\in V$:

$$\begin{array}{ll}
 & f\cdot v \\
= & \quad \{ \text{ property of } \varphi \ \} \\
 & f\cdot(\varphi\cdot v) \\
= & \quad \{ \text{ definition (2) of } f \text{, using } b\cdot(\varphi\cdot v) \ \} \\
 & h\cdot(\varphi\cdot v) \ \triangleright\ f\cdot(k\cdot(\varphi\cdot v)) \\
= & \quad \{ \text{ introduce functions } hh \text{ and } kk \ \} \\
 & hh\cdot v \ \triangleright\ f\cdot(kk\cdot v) \quad .
\end{array}$$

Functions hh and kk are just the compositions of h and k with φ:

$$hh = h\circ\varphi \ \text{ and: } \ kk = k\circ\varphi \quad .$$

Thus we obtain that function f, as defined by (2), satisfies:

(4) $\quad f\cdot v \ = \ hh\cdot v \ \triangleright\ f\cdot(kk\cdot v)$, for all $v\in V$.

This is a recursive definition for f and we have shown that every solution of (2) is a solution of (4) as well. Because the latter is uniformly productive, it uniquely defines f as a function of type $V\to\mathcal{L}_\infty(B)$; therefore, the original definition (2) also uniquely defines f as a function of this type.

10.7 Degrees of Productivity

As we have seen, functions of the shape $(b\triangleright)$ are productive whereas a function like $(\lfloor 1)$ certainly is not productive. The composition of such a non-productive function with other, "sufficiently productive", functions may nevertheless yield a productive function again. For example:

$$(b\triangleright)\circ(c\triangleright)\circ(\lfloor 1) \quad ,$$

is productive because the lack of productivity of $(\lfloor 1)$ is sufficiently compensated for by $(b\triangleright)$ and $(c\triangleright)$.

Hence, to express the productivity of a composition of some functions in terms of their individual productivity we must generalize the notion of productivity. The relevance of this is that many productive functions can be considered as compositions of standard functions. By establishing the (degrees of) productivity of these standard functions once and for all, the (degree of) productivity of their compositions can be established easily, without further reference to the (somewhat awkward) definition of productivity.

The key formula in the definition of productivity was:

$$(\forall x,j :: x\in\mathcal{P}_j \Rightarrow F\cdot x\in\mathcal{P}_{j+1}) \quad .$$

The generalization now is that function $(+1)$, as present in $\mathcal{P}_{j+1}$, is now replaced by any function f, of type $\mathsf{Int}\to\mathsf{Int}$; we call this *$f$-productivity*. Our original notion of productivity then becomes $(+1)$-productivity.

definition 15 (General Productivity): For function f, of type $\mathsf{Int}\to\mathsf{Int}$, a function F, of type $\Omega\to\Omega$, is f-productive if it has the following two properties:

(α) $\quad (\forall x,j :: x\in\mathcal{P}_j \Rightarrow F\cdot x\in\mathcal{P}_{f\cdot j})$

(β) $\quad (\forall x,y,j : x,y\in\mathcal{L}_\infty : x\lceil j = y\lceil j \;\Rightarrow\; F\cdot x\lceil f\cdot j = F\cdot y\lceil f\cdot j)$

□

The relation with the original notion of productivity is given formally by the following property.

property: If $(\forall j :: f\cdot j > j)$ then f-productivity implies productivity.
□

To all practical intents and purposes, we can restrict ourselves to integer functions f that are *ascending*, and we will do so tacitly.

examples: For any b, natural j, and function f:

$(b\triangleright)$ is $(+1)$-productive
I is $(+0)$-productive
$(f\bullet)$ is $(+0)$-productive
$(\lfloor j)$ is $(-j)$-productive

□

Now we can formulate the following Composition Rule for productivity of composite functions. The requirement of ascendingness is necessary for its proof. Fortunately, if f and g are ascending then so is $f\circ g$.

theorem 16 (Composition Rule): For ascending integer functions f and g, and for functions F and G:

if F is f-productive and G is g-productive

then $F\circ G$ is $f\circ g$-productive

□

10.8 Examples and Exercises

Infinite lists and functions yielding infinite lists are always defined recursively. The functions of which they are fixed points usually remain anonymous. If this anonymous function is productive we also call the definition itself (of that function or infinite list) *productive*.

Here we present a few simple examples of productive definitions, whereas in the next chapter we illustrate how such definitions can be derived. Properties can be proved in two ways: either by an appeal to the Second Productivity Theorems for admissible predicates, or directly by induction on the domain of the lists.

10.8.0 map

For function f of type $B\to C$, function $(f\bullet)$ has type $\mathcal{L}_\infty(B)\to\mathcal{L}_\infty(C)$; we recall[0] its recursive definition, for parameters of the shape $b\triangleright x$:

[0] from Chapter 6

$$f\bullet(b\triangleright x) \;=\; f\cdot b \triangleright f\bullet x \quad .$$

This definition is uniformly productive. Thus defined, $f\bullet x$ satisfies, for all $x\in\mathcal{L}_\infty$:

$$(\forall i :: (f\bullet x)\cdot i = f\cdot x\cdot i) \quad .$$

This can be proved by means of Theorem 14, or directly from the definition, by induction on i.

In addition, for any (fixed) function f we have that $(f\bullet)$, viewed as a function of type $\Omega\to\Omega$, is $(+0)$-productive. By combining $(f\bullet)$ with $(+1)$-productive functions we obtain productive functions that may be useful. For instance – see the next chapter for the details –:

$$x \;=\; b \triangleright f\bullet x$$

defines x as the list with $x\cdot i = f^i\cdot b$, for all i; an instance of this is:

$$nats \;=\; 0 \triangleright (+1)\bullet nats \quad ,$$

which defines $nats$ as the list of natural numbers. Another special case is:

$$pows \;=\; 1 \triangleright (2*)\bullet pows \quad ,$$

which defines $pows$ as the list of all powers of two: $pows\cdot i = 2^i$, for all i.

10.8.1 sums of successive pairs

Suppose function F is defined recursively by:

(5) $$F\cdot(b\triangleright c\triangleright x) \;=\; (b+c) \triangleright F\cdot(c\triangleright x) \quad .$$

This definition is uniformly productive and defines F as a function of type $\mathcal{L}_\infty(\mathsf{Int})\to\mathcal{L}_\infty(\mathsf{Int})$. In addition, F satisfies, for any infinite integer list x:

$$(\forall i :: F\cdot x\cdot i = x\cdot i + x\cdot(i+1)) \quad .$$

10.8.2 the Fibonacci numbers

Function F, as defined by (5) above, when viewed as a function of type $\Omega\to\Omega$, is (-1)-productive. We now define function G thus:

(6) $$G\cdot x \;=\; 0 \triangleright 1 \triangleright F\cdot x \quad .$$

So $G = (0\triangleright)\circ(1\triangleright)\circ F$ and G is $(+1)$-productive. Hence, it has a unique fixed foint which is an infinite list. In the next chapter we will derive definitions (5) and (6) from a specification of the infinite list of Fibonacci numbers: this turns out to be G's fixed point.

10.8.3 exercises

0. Prove the equivalence of definitions 1 and 2.

1. Prove Property 4.

2. Prove Property 7.

3. Formulate a definition for the predicate *inc* that expresses increasingness of infinite lists of numbers. Prove that *inc* is admissible.

4. Prove part (β) of the First Uniform Productivity Theorem.

5. Prove lemma 13.

6. With F and G as defined in (5) and (6), infinite list s is defined by $s = G\cdot s$. Prove $(\forall i :: s\cdot i = \mathit{fib}\cdot i)$.

7. For given b and f and x defined recursively by $x = b \triangleright f\bullet x$, prove: $(\forall i :: x\cdot i = f^i\cdot b)$, both by using Theorem 14 and by direct mathematical induction.

8. For functions f, of type $\mathsf{Nat}\to V$, derive a declaration for a function *lst*, of type $(\mathsf{Nat}\to V)\to\mathcal{L}_\infty(V)$, and satisfying $(\forall f, i :: \mathit{lst}\cdot f\cdot i = f\cdot i)$.

9. Function F is declared by $F\cdot(b\triangleright x) = b\triangleright x\lfloor 1$, whereas G is declared by $G\cdot(b\triangleright c\triangleright x) = b\triangleright x$. Do or do not F and G have the same degrees of productivity?

10. Functions *dup* and *half* are defined by $\mathit{dup}\cdot(b\triangleright x) = b\triangleright b\triangleright \mathit{dup}\cdot x$ and $\mathit{half}\cdot(b\triangleright x) = b\triangleright \mathit{half}\cdot(x\lfloor 1)$. What are the degrees of productivity of these functions? Show that both $\mathit{dup}\circ\mathit{half}$ and $\mathit{half}\circ\mathit{dup}$ are $(+0)$-productive.

Chapter 11

Programming with Infinite Lists

In this chapter we illustrate a few techniques for constructing declarations of (functions yielding) infinite lists from given specifications. We start with a few very simple examples, to show that, generally, several approaches are possible.

11.0 Enumerating the Naturals

The natural numbers can be enumerated in an infinite list. This means that we are interested in an infinite list $nats$, say, satisfying:

$$(0) \qquad (\forall i :: nats \cdot i = i) \quad .$$

Viewed as a function, $nats$ just is the identity function in $\mathsf{Nat} \to \mathsf{Nat}$, represented as an infinite list. To obtain such infinite list we must somehow introduce operator $\triangleright$; we do so by means of the $\triangleright$-trick we have used earlier:

$$\begin{array}{ll} & nats \cdot 0 \\ = & \quad \{ \text{ specification (0) of } nats \text{ } \} \\ & 0 \\ = & \quad \{ \text{ property of } \triangleright \text{ } \} \\ & (0 \triangleright ?) \cdot 0 \quad , \end{array}$$

from which we conclude that we are looking for a declaration of the shape $nats = 0 \triangleright ?$. Furthermore, for all natural i:

$$
\begin{array}{ll}
 & nats\cdot(i{+}1) \\
= & \quad \{ \text{ specification (0) of } nats \text{ } \} \\
 & i+1 \\
= & \quad \{ \text{ specification (0) of } nats\text{, by Induction Hypothesis } \} \\
 & nats\cdot i+1 \\
= & \quad \{ \text{ sectioning, to isolate function } (+1) \text{ } \} \\
 & (+1)\cdot(nats\cdot i) \\
= & \quad \{ \text{ } nats \text{ is a list: property of } \bullet \text{ } \} \\
 & ((+1)\bullet nats)\cdot i \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (\,?\,\triangleright\,(+1)\bullet nats\,)\cdot(i{+}1) \quad .
\end{array}
$$

Combining the results of these derivations we obtain as declaration for $nats$:

$$nats \;=\; 0 \triangleright (+1)\bullet nats \quad .$$

This definition is productive; hence, $nats$ is an infinite list indeed.

The purpose of the steps "sectioning" and "property of $\bullet$" is to transform the expression into one of the shape $(\cdots)\cdot i$, because we know how to solve $nats$ from an equation of the shape $nats\cdot i = (\cdots)\cdot i$. In other words, the purpose of these steps is to abstract from index i.

* * *

Another approach to the same effect runs as follows. Having decided that we are heading for a declaration of the shape $nats = 0 \triangleright ?$, we may as well introduce a name x, say, for the missing part. So, we are now heading for a declaration of the shape $nats = 0 \triangleright x$, and we investigate what properties x should have. Expression $0 \triangleright x$ is an infinite list if (and only if) x is an infinite list; hence, x must be an infinite list too, and $nats$'s specification must be satisfied as well:

$$
\begin{array}{ll}
 & x\cdot i \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (0 \triangleright x)\cdot(i{+}1) \\
= & \quad \{ \text{ proposed definition of } nats \text{ } \}
\end{array}
$$

$$
\begin{array}{ll}
 & nats\cdot(i{+}1) \\
= & \quad \{ \text{ specification (0) of } nats \text{ } \} \\
 & i+1 \quad ,
\end{array}
$$

from which we conclude that $nats$'s specification is satisfied provided that x satisfies:

$$(\forall i :: x\cdot i = i+1) \quad .$$

We can now apply the same transformations as above, but we also observe that i and $i{+}1$ are instances of the more general expression $i{+}j$. Thus, we consider both $nats$ and x as instances of a more general function $from$, of type $\mathsf{Nat}\to\mathcal{L}_\infty(\mathsf{Nat})$, and with specification:

(1) $\quad (\forall j, i :: from\cdot j\cdot i = i+j) \quad .$

With this specification $nats$ can be declared in terms of $from$ as follows:

$$nats \;=\; from\cdot 0 \quad .$$

A declaration for $from$ can now be derived easily, by induction on i; that is, we prove $(\forall i :: (\forall j :: from\cdot j\cdot i = i+j))$ by Mathematical Induction:

$$
\begin{array}{ll}
 & from\cdot j\cdot 0 \\
= & \quad \{ \text{ specification (1) of } from \text{ } \} \\
 & 0+j \\
= & \quad \{ \text{ algebra } \} \\
 & j \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (j \triangleright ?)\cdot 0 \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & from\cdot j\cdot(i{+}1) \\
= & \quad \{ \text{ specification (1) of } from \text{ } \} \\
 & (i+1) + j \\
= & \quad \{ \text{ algebra } \} \\
 & i + (j+1) \\
= & \quad \{ \text{ specification (1) of } from\text{, by Induction Hypothesis } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{from}\cdot(j{+}1)\cdot i \\
= & \quad \{ \text{ property of } \triangleright \ \} \\
 & (?\triangleright \mathit{from}\cdot(j{+}1))\cdot(i{+}1) \quad .
\end{array}
$$

Thus we obtain:

$$\mathit{from}\cdot j \;=\; j \triangleright \mathit{from}\cdot(j{+}1) \quad .$$

This definition is uniformly productive, so $\mathit{from}\cdot j$ is an infinite list, for all j.

11.1 Function Listification

For every function f, of type $\mathsf{Nat}\to V$, an infinite list x, in $\mathcal{L}_\infty(V)$, exists that is functionally equivalent to f; that is, a specification for x is:

(2) $\quad x\in\mathcal{L}_\infty \;\wedge\; (\forall i :: x\cdot i = f\cdot i) \quad .$

A declaration for such a list can be obtained in several ways, a few of which we investigate.

11.1.0 map over the naturals

The more difficult half of x's specification is the conjunct $x\in\mathcal{L}_\infty$; with this in mind we derive:

$$
\begin{array}{ll}
 & x\cdot i \\
= & \quad \{ \text{ specification (2) of } x \ \} \\
 & f\cdot i \\
= & \quad \{ \text{ specification (0) of } \mathit{nats}\text{, to introduce a list } \} \\
 & f\cdot(\mathit{nats}\cdot i) \\
= & \quad \{ \ \mathit{nats} \text{ is an infinite list: property of } \bullet \ \} \\
 & (f\bullet \mathit{nats})\cdot i \quad .
\end{array}
$$

Because nats is an infinite list, $f\bullet\mathit{nats}$ is an infinite list as well; thus we conclude that:

$$x \;=\; f\bullet\mathit{nats}$$

properly declares x as the infinite list representing f.

By turning f into a parameter we can also define a generic function lst that maps any function in $\mathsf{Nat} \to V$ to a functionally equivalent infinite list in $\mathcal{L}_\infty(V)$:

$$lst \cdot f \;=\; f \bullet nats \quad .$$

11.1.1 by means of a function

We also can decide right away to introduce a function lst, mapping a function of type $\mathsf{Nat} \to V$ to $\mathcal{L}_\infty(V)$ and with this specification:

(3) $\quad (\forall i :: lst \cdot f \cdot i = f \cdot i) \quad$, for all functions f .

Now we derive, using induction on i:

$$\begin{array}{ll} & lst \cdot f \cdot 0 \\ = & \quad \{ \text{ specification (3) of } lst \ \} \\ & f \cdot 0 \\ = & \quad \{ \text{ property of } \triangleright \ \} \\ & (f \cdot 0 \triangleright ?) \cdot 0 \quad , \end{array}$$

and:

$$\begin{array}{ll} & lst \cdot f \cdot (i{+}1) \\ = & \quad \{ \text{ specification (3) of } lst \ \} \\ & f \cdot (i{+}1) \quad , \end{array}$$

and from this point we can proceed in several directions: this situation is similar to the special case for $nats$.

One option is to rewrite $f \cdot (i{+}1)$ to $(f \circ (+1)) \cdot i$ and apply lst recursively to function $f \circ (+1)$, which leads to this solution for lst:

$$lst \cdot f \;=\; f \cdot 0 \triangleright lst \cdot (f \circ (+1)) \quad .$$

This definition is, however, not very efficient, because evaluation of $lst \cdot f$ will give rise to a lot of continued function compositions, one for each unfolding of an application of lst. As these function compositions all pertain to one and the same function, namely $(+1)$, a more efficient solution must be possible, in which these continued compositions are represented more efficiently.

Another possibility is to observe, again, that $f \cdot i$ and $f \cdot (i{+}1)$ are instances of the more general expression $f \cdot (i{+}j)$. Therefore, we generalize *lst* to a function *glst*, specified by:

(4) $\quad (\forall j, i :: \mathit{glst} \cdot f \cdot j \cdot i = f \cdot (i{+}j))$, for function f .

In addition, of course, $\mathit{glst} \cdot f \cdot j$ must be an infinite list. Note that, actually, $\mathit{glst} \cdot f \cdot j$ implements $\mathit{lst} \cdot (f \circ (+j))$, and that function $(+j)$ equals the continued composition $(+1)^j$.

In terms of *glst* a declaration for *lst* is:

$$\mathit{lst} \cdot f \;=\; \mathit{glst} \cdot f \cdot 0 \quad .$$

The derivation of a declaration for *glst* is not difficult either; as before, we employ induction on i:

$$\begin{array}{ll} & \mathit{glst} \cdot f \cdot j \cdot 0 \\ = & \quad \{ \text{ specification (4) of } \mathit{glst} \text{ ; algebra } \} \\ & f \cdot j \quad , \end{array}$$

and:

$$\begin{array}{ll} & \mathit{glst} \cdot f \cdot j \cdot (i{+}1) \\ = & \quad \{ \text{ specification of } \mathit{glst} \text{ } \} \\ & f \cdot ((i{+}1){+}j) \\ = & \quad \{ \text{ algebra } \} \\ & f \cdot (i{+}(j{+}1)) \\ = & \quad \{ \text{ specification (4) of } \mathit{glst} \text{ , by Induction Hypothesis } \} \\ & \mathit{glst} \cdot f \cdot (j{+}1) \cdot i \quad . \end{array}$$

Thus we obtain, by combination of the two cases and abstraction from i:

$$\mathit{glst} \cdot f \cdot j \;=\; f \cdot j \triangleright \mathit{glst} \cdot f \cdot (j{+}1) \quad .$$

This definition is uniformly productive, hence $\mathit{glst} \cdot f \cdot j$ is an infinite list, as required.

11.2 The Fibonacci Numbers

Function *lst* provides a general-purpose way to construct an infinite list representing any function on Nat. Calculation of the first n elements of this list requires n independent evaluations of the function involved. This may turn out to be inefficient, particularly so if an efficient recurrence relation exists for the function: this recurrence relation will not be exploited by *lst*. In such cases, ad-hoc derivations may yield more efficient solutions.

As an example, we derive an efficient list for the Fibonacci function *fib*, defined by, for natural i:

$$\begin{array}{rcl} \mathit{fib}\cdot 0 & = & 0 \\ \mathit{fib}\cdot 1 & = & 1 \\ \mathit{fib}\cdot(i{+}2) & = & \mathit{fib}\cdot i + \mathit{fib}\cdot(i{+}1) \end{array}$$

An infinite list for function *fib* is the infinite list *fbs*, with specification:

(5) $\quad (\forall i :: \mathit{fbs}\cdot i = \mathit{fib}\cdot i) \quad .$

In view of the case analysis in *fib*'s definition, we derive:

$$\begin{array}{ll} & \mathit{fbs}\cdot 0 \\ = & \quad \{ \text{ specification (5) of } \mathit{fbs} \text{ } \} \\ & \mathit{fib}\cdot 0 \\ = & \quad \{ \text{ definition of } \mathit{fib} \text{ } \} \\ & 0 \\ = & \quad \{ \text{ property of } \triangleright \text{ } \} \\ & (0 \triangleright ?)\cdot 0 \quad , \end{array}$$

and, very similarly:

$$\begin{array}{ll} & \mathit{fbs}\cdot 1 \\ = & \quad \{ \text{ specification (5) of } \mathit{fbs} \text{ and definition of } \mathit{fib} \text{ } \} \\ & 1 \\ = & \quad \{ \text{ property of } \triangleright \text{ (twice) } \} \\ & (? \triangleright 1 \triangleright ?)\cdot 1 \quad . \end{array}$$

Also, for all natural i:

$$
\begin{array}{ll}
 & \mathit{fbs}\cdot(i{+}2) \\
= & \quad \{ \text{ specification (5) of } \mathit{fbs} \ \} \\
 & \mathit{fib}\cdot(i{+}2) \\
= & \quad \{ \text{ definition of } \mathit{fib} \ \} \\
 & \mathit{fib}\cdot i \ + \ \mathit{fib}\cdot(i{+}1) \\
= & \quad \{ \text{ specification (5) of } \mathit{fbs}\text{, by Induction Hypothesis } \} \\
 & \mathit{fbs}\cdot i \ + \ \mathit{fbs}\cdot(i{+}1) \\
= & \quad \{ \text{ new function } \mathit{ps}\text{, to isolate } i\text{, see below } \} \\
 & \mathit{ps}\cdot\mathit{fbs}\cdot i \\
= & \quad \{ \text{ property of } \triangleright \text{ (twice) } \} \\
 & (?\triangleright ?\triangleright \mathit{ps}\cdot\mathit{fbs})\cdot(i{+}2) \quad .
\end{array}
$$

The only way to transform expression $\mathit{fbs}\cdot i \ + \ \mathit{fbs}\cdot(i{+}1)$ into an expression of the shape $(\cdots)\cdot i$ is to introduce a new function just for this purpose. Here we have introduced a new function ps, that has type $\mathcal{L}_\infty(\mathsf{Int}) \to \mathcal{L}_\infty(\mathsf{Int})$ and, for the sake of the correctness of its use in the above derivation, it suffices if ps has this specification, for any infinite integer list x:

$$
(6) \qquad (\forall i :: \mathit{ps}\cdot x\cdot i \ = \ x\cdot i + x\cdot(i{+}1)\,) \quad .
$$

Combining the results of the above three derivations we then obtain as solution for the list of Fibonacci numbers (in terms of our new function ps):

$$
\mathit{fbs} \ = \ 0 \triangleright 1 \triangleright \mathit{ps}\cdot\mathit{fbs} \quad .
$$

Thus we are left with the obligation to construct a suitable declaration for ps; to this end we derive, by induction on i:

$$
\begin{array}{ll}
 & \mathit{ps}\cdot x\cdot 0 \\
= & \quad \{ \text{ specification (6) of } \mathit{ps} \ \} \\
 & x\cdot 0 \ + \ x\cdot 1 \\
= & \quad \{ \text{ property of } \triangleright \ \} \\
 & (\,(x\cdot 0 + x\cdot 1) \triangleright ?\,)\cdot 0 \quad ,
\end{array}
$$

and:

$$
\begin{array}{ll}
 & \mathit{ps}\cdot x\cdot(i{+}1) \\
= & \quad \{ \text{ specification (6) of } \mathit{ps} \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & x\cdot(i{+}1) \;+\; x\cdot(i{+}2) \\
= & \quad \{ \text{ property of } \lfloor 1 \text{ , to decrease the indices } \} \\
 & (x\lfloor 1)\cdot i \;+\; (x\lfloor 1)\cdot(i{+}1) \\
= & \quad \{ \text{ specification (6) of } ps\text{, by Induction Hypothesis } \} \\
 & ps\cdot(x\lfloor 1)\cdot i \\
= & \quad \{ \text{ property of } \triangleright \} \\
 & (\,?\triangleright ps\cdot(x\lfloor 1)\,)\cdot(i{+}1) \quad .
\end{array}
$$

Thus we obtain as declaration for ps:

$$ps\cdot x \;=\; (x\cdot 0{+}x\cdot 1) \triangleright ps\cdot(x\lfloor 1) \quad ,$$

which can be reformulated, with parameter patterns, as:

$$ps\cdot(b\triangleright c\triangleright y) \;=\; (b{+}c) \triangleright ps\cdot(c\triangleright y) \quad .$$

Summarizing, we obtain this declaration for the infinite list of Fibonacci numbers:

$$
\begin{array}{rrcl}
 & fbs & = & 0 \triangleright 1 \triangleright ps\cdot fbs \\
\& & ps\cdot(b\triangleright c\triangleright y) & = & (b{+}c) \triangleright ps\cdot(c\triangleright y)
\end{array}
$$

11.3 The Sums of the Initial Segments

11.3.0 the problem

In the previous section we have seen a function, namely ps, mapping infinite lists to infinite lists. The derivation of a declaration for this function was a rather straightforward affair.

Specifications of functions from infinite lists to infinite lists often admit several approaches, however, of which the more obvious ones do not always lead to the most efficient solutions.

To illustrate this we consider the following example. Function f is required to have type $\mathcal{L}_\infty(\mathsf{Int})\to\mathcal{L}_\infty(\mathsf{Int})$ and is specified by:

(7) $\quad (\forall s, j :: f\cdot s\cdot j = sum\cdot(s\lceil j)\,) \quad ,$

where function sum maps finite integer lists to the sums of their elements.

In words, this specification states that element j of list $f\cdot s$ is the sum of list s's first j elements. As we will see, this problem admits of (at least) two solutions, of which, however, the less obvious one turns out to be the more efficient one.

11.3.1 first solution

We use Mathematical Induction on j, that is, we solve f from

(8) $\quad (\forall j :: (\forall s :: f\cdot s\cdot j = sum\cdot(s\lceil j)\,)\,)$

by using Mathematical Induction on j. We derive:

$$
\begin{array}{ll}
 & f\cdot s\cdot 0 \\
= & \quad \{ \text{ specification (8) of } f \text{ } \} \\
 & sum\cdot(s\lceil 0) \\
= & \quad \{ \text{ property of } \lceil \text{ } \} \\
 & sum\cdot([\,]) \\
= & \quad \{ \text{ property of } sum \text{ } \} \\
 & 0 \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (0 \triangleright ?)\cdot 0 \quad ,
\end{array}
$$

which indicates a definition for f of the shape $f\cdot s = 0 \triangleright ?$. Furthermore, we derive, for any natural j:

$$
\begin{array}{ll}
 & f\cdot s\cdot(j{+}1) \\
= & \quad \{ \text{ specification (8) of } f \text{ } \} \\
 & sum\cdot(s\lceil(j{+}1)) \\
= & \quad \{ \text{ property of } \lceil \text{ } \} \\
 & sum\cdot(\,s\cdot 0 \triangleright s\lfloor 1\lceil j\,) \\
= & \quad \{ \text{ property of } sum \text{ } \} \\
 & s\cdot 0 + sum\cdot(s\lfloor 1\lceil j) \\
= & \quad \{ \text{ specification (8) of } f\text{, by Ind. Hyp., with } s := s\lfloor 1 \text{ } \} \\
 & s\cdot 0 + f\cdot(s\lfloor 1)\cdot j \\
= & \quad \{ \text{ sectioning and } \bullet \text{ , to isolate } j \text{ } \} \\
 & (\,(s\cdot 0{+})\bullet f\cdot(s\lfloor 1)\,)\cdot j \\
= & \quad \{ \text{ property of } \triangleright \text{ } \} \\
 & (\,? \triangleright (s\cdot 0{+})\bullet f\cdot(s\lfloor 1)\,)\cdot(j{+}1) \quad .
\end{array}
$$

By combining the results of these two derivations we obtain as recursive declaration for f:

$$f \cdot s \;=\; 0 \triangleright (s \cdot 0+) \bullet f \cdot (s \lfloor 1) \quad ,$$

which, by means of parameter patterns, can also be written as:

$$f \cdot (b \triangleright t) \;=\; 0 \triangleright (b+) \bullet f \cdot t \quad . \tag{9}$$

This solution has quadratic time complexity: evaluation of $(f \cdot s) \lceil n$ will require $\mathcal{O}(n^2)$ steps.

In the above derivation we have used this property of $\lceil$:

$$s \lceil (j+1) \;=\; s \cdot 0 \triangleright s \lfloor 1 \lceil j \quad ,$$

which corresponds to the "usual" head-tail decomposition of a non-empty finite list. In this respect this solution is the more obvious one, but we can, however, do better.

11.3.2 second solution

Again, we use Mathematical Induction on j. The derivation for the base case remains the same, and for any natural j we derive, now using a different property of $\lceil$:

$$\begin{array}{ll}
 & f \cdot s \cdot (j+1) \\
= & \quad \{ \text{ specification (8) of } f \ \} \\
 & sum \cdot (s \lceil (j+1)) \\
= & \quad \{ \text{ property of } \lceil \text{, now in terms of } \triangleleft \ \} \\
 & sum \cdot (s \lceil j \triangleleft s \cdot j) \\
= & \quad \{ \text{ property of } sum \ \} \\
 & sum \cdot (s \lceil j) + s \cdot j \\
= & \quad \{ \text{ specification (8) of } f \text{, by Induction Hypothesis } \} \\
 & f \cdot s \cdot j + s \cdot j \\
= & \quad \{ \text{ new function } add \text{, to isolate } j \text{, see below } \} \\
 & add \cdot (f \cdot s) \cdot s \cdot j \\
= & \quad \{ \text{ property of } \triangleright \ \} \\
 & (\,? \triangleright add \cdot (f \cdot s) \cdot s\,) \cdot (j+1) \quad .
\end{array}$$

Here we have used this property of $\lceil$:

$$s\lceil(j{+}1) \;=\; s\lceil j \lhd s\cdot j \quad ,$$

which may seem a less obvious thing to do, but this, all by itself, does not make it less valuable.

By combining this result with the earlier result for the base case we now obtain as recursive declaration for f:

$$(10) \qquad f\cdot s \;=\; 0 \rhd add\cdot(f\cdot s)\cdot s \quad .$$

Function add maps two infinite lists (of integers) to the infinite list of their pairwise sums; that is, its specification is, for infinite lists s and t:

$$(\forall j :: add\cdot t\cdot s\cdot j \;=\; t\cdot j + s\cdot j) \quad .$$

Deriving a declaration for add is straightforward (and similiar to the derivation for ps in the previous section). This leads to:

$$add\cdot(c\rhd t)\cdot(b\rhd s) \;=\; (c{+}b) \rhd add\cdot t\cdot s \quad .$$

Declaration (10) also has quadratic time complexity, but it allows a simple transformation that makes the time complexity linear. The crucial observation is that, in the defining expression of (10), function f is applied recursively to its own parameter, that is, to the *same* list. Hence, this parameter appears in the definition (and in the computation) as a *constant* and, as a consequence, the value $f\cdot s$ appears as a constant as well; therefore, this value can be *named* and shared:

$$f\cdot s \;=\; x \;\mathsf{whr}\; x = 0 \rhd add\cdot x\cdot s \;\mathsf{end} \quad .$$

This, at first sight seemingly innocent, transformation yields a solution with linear time complexity. Therefore, this approach deserves to be remembered. In summary, the technique is: try to obtain a pattern of recursion in which the function's value is defined recursively in terms of itself, instead of a recursive application of the *same* function to a *different* argument. By subsequently giving that value a name its repeated evaluation is precluded and this avoidance of repeated evaluation of the same value yields the improvement of efficiency.

11.4 Enumerating Sets of Naturals

We consider infinite sets of natural numbers and their representation by infinite lists. To make the representation unique, we use *increasing* infinite lists. To begin with, we introduce some additional notation.

11.4.0 lists as representations of sets

Every (infinite) *list* of naturals represents an (infinite) *set* of naturals, which is obtained by abstraction from the order of the elements in the list. We denote the abstraction function by a bracket pair $[\![\cdot]\!]$, and for any infinite list x we define the set of its elements, $[\![x]\!]$, as follows:

$$[\![x]\!] \;=\; (\mathsf{set}\, i : 0 \leq i : x \cdot i) \quad .$$

So, $[\![x]\!]$ is just the set of all values occurring in x, without regard for their order. A rather obvious property of $[\![\cdot]\!]$ is, of course:

$$[\![b \triangleright x]\!] \;=\; \{b\} \cup [\![x]\!] \quad .$$

The relation "being an element of", on sets denoted by $\in$, can now be defined on lists as well, for which we use the same symbol $\in$:

$$b \in x \;\equiv\; b \in [\![x]\!] \quad , \text{ for all } b \text{ and list } x \quad .$$

Obvious properties of this relation are, for any b, c, and x:

$$b \in x \;\equiv\; (\exists i : 0 \leq i : b = x \cdot i) \quad , \text{ and:}$$

$$b \in (c \triangleright x) \;\equiv\; b = c \;\vee\; b \in x \quad .$$

In addition, every function f satisfies, for every infinite list x:

(11) $$f \bullet [\![x]\!] \;=\; [\![f \bullet x]\!] \quad ,$$

which means that $\bullet$ commutes with $[\![\cdot]\!]$.

11.4.1 increasing lists

An infinite list of (any kind of) numbers is increasing if it satisfies predicate inc, defined by, for infinite list x of numbers:

$$inc \cdot x \;\equiv\; (\forall i,j : 0 \le i < j : x \cdot i < x \cdot j) \quad .$$

By distinguishing the cases $0 = i$ and $1 \le i$, and by substituting $b \triangleright x$ for x we can rewrite this as:

$$inc \cdot (b \triangleright x) \;\equiv\; (\forall j : 0 \le j : b < x \cdot j) \;\wedge\; (\forall i,j : 0 \le i < j : x \cdot i < x \cdot j) \quad .$$

Using relation $\in$ and using inc recursively, we obtain this, nice and concise, property of inc:

$$(12) \qquad inc \cdot (b \triangleright x) \;\equiv\; (\forall c : c \in [\![x]\!] : b < c) \;\wedge\; inc \cdot x \quad .$$

The first conjunct expresses that b is the *minimum* of list $b \triangleright x$, whereas the second conjunct expresses that the tail, x, of list $b \triangleright x$ is increasing: an infinite list is increasing if and only if its head is its minimum and its tail is increasing.

Predicate inc is admissible and formula (12) turns out to be convenient because it matches the proof patterns of the Productivity Theorems very well. Notice, however, that formula (12) does not fully *define* predicate inc: the constant predicate false has this property too; hence, (12) is not a sufficent definition, it is just a property, albeit a very useful one.

Because of the conjunct $inc \cdot x$, in property (12), the minimum of list x is $x \cdot 0$; therefore, property (12) can also be formulated as follows:

$$(13) \qquad inc \cdot (b \triangleright x) \;\equiv\; b < x \cdot 0 \;\wedge\; inc \cdot x \quad ,$$

which is formally weaker than (12) and which, therefore, may be easier as a proof obligation.

* * *

One and the same set can be represented by many different lists, because the order in which the set's elements occur in the list is irrelevant. The representation of a set of naturals by an *increasing* infinite list, however, is unique. This is the essence of the following lemma.

lemma "uni": For infinite integer lists x and y:

$$inc \cdot x \ \wedge \ inc \cdot y \ \Rightarrow \ (\, [\![\, x \,]\!] = [\![\, y \,]\!] \ \equiv \ x = y \,) \quad .$$

□

Notice that the implication $[\![\, x \,]\!] = [\![\, y \,]\!] \Leftarrow x = y$ does not require the lists to be increasing: this is just an instance of Leibniz's rule. The lemma's mathematical content is in the other implication, $[\![\, x \,]\!] = [\![\, y \,]\!] \Rightarrow x = y$; the use of "$\equiv$" in the lemma's formulation just makes it formally stronger and, hence, somewhat more easily applicable.

11.4.2 infinite sets as lists

Every infinite subset U of Nat can be represented by an increasing infinite list x, specified by:

(14) $\quad [\![\, x \,]\!] = U \ \wedge \ inc \cdot x \quad .$

We show that such list exists, by constructing a definition for a function lst mapping all infinite subsets U of Nat to infinite lists of naturals satisfying specification (14).

For the time being we will ignore the problem how to implement the operations on infinite subsets of Nat. In any particular application this issue must be considered, of course, but here we ignore it, because we are mainly interested in the approach and the general structure of the solution, rather than the details of its implementation.

Because its value has to be an infinite list, $lst \cdot U$ will be an expression of the shape $b \triangleright y$. Because the head of an increasing list is its minimum we can only take the minimum of U for b, and y must contain all other – than that minimum – elements of U. So, without much trouble we invent this proposal for a recursive definition for lst:

(15) $\quad lst \cdot U \ = \ b \triangleright lst \cdot (U \backslash \{\, b \,\}) \ \ \mathsf{whr} \ \ b = \ \mathsf{min} \cdot U \ \mathsf{end} \quad ,$

but we still must prove that this definition indeed satisfies specification (14). (Note that $\mathsf{min} \cdot U$ exists because every non-empty subset of the naturals has a minimum.) To this end we have to prove the following three properties, for all infinite U, $U \subseteq \mathsf{Nat}$:

(16) $\quad lst \cdot U \in \mathcal{L}_\infty(\mathsf{Nat})$

(17) $\quad [\![\, lst \cdot U \,]\!] = U$

(18) $\quad inc \cdot (lst \cdot U)$

To prepare for application of the theorems for (uniform) productivity, from the previous chapter, we observe that (definition (15) is equivalent to):

(19) $\quad lst \;=\; F\cdot lst$, where

function F is defined by:

(20) $\quad F\cdot f\cdot U \;=\; b \triangleright f\cdot(U\backslash\{b\})$ whr $b = \mathsf{min}\cdot U$ end .

The shape of this definition directly matches the pattern required for application of (First Uniform Productivity) Theorem 9 – in the previous chapter – ; as a result, $lst\cdot U$ is an infinite list, for every infinite set U of naturals. Hence, this proves property (16). Notice that this requires $U\backslash\{b\}$ to be an infinite set of naturals too, which is the case if (and only if) U is infinite itself.

Property (17) is the conjunction of the following two weaker properties, which we will prove separately – for all U – :

(21) $\quad [\![\, lst\cdot U \,]\!] \subseteq U$

(22) $\quad U \subseteq [\![\, lst\cdot U \,]\!]$

To prove property (21) we define a predicate P_0 , as follows, for all functions f mapping infinite subsets of Nat to $\mathcal{L}_\infty(\mathsf{Nat})$:

(23) $\quad P_0\cdot f \;\equiv\; (\forall U :: [\![\, f\cdot U \,]\!] \subseteq U)$.

Property (21) now amounts to $P_0\cdot lst$, which follows from (Second Uniform Productivity) Theorem 14, provided we prove that predicate P_0 is uniformly admissible and that P_0 satisfies the two premisses of this theorem.

The following calculation shows that predicate P_0 is uniformly admissible:

$$
\begin{array}{ll}
 & P_0\cdot f \\
\equiv & \quad \{ \text{ definition (23) of } P_0 \text{ } \} \\
 & (\forall U :: [\![\, f\cdot U \,]\!] \subseteq U) \\
\equiv & \quad \{ \text{ definition of } [\![\cdot]\!] \text{ } \} \\
 & (\forall U :: (\mathsf{set}\, i : 0 \leq i : f\cdot U\cdot i) \subseteq U) \\
\equiv & \quad \{ \text{ definitions of } \subseteq \text{ and } \mathsf{set} \text{ } \} \\
 & (\forall U :: (\forall i : 0 \leq i : f\cdot U\cdot i \in U)) \\
\equiv & \quad \{ \text{ predicate calculus } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall U :: (\forall j : 0 \leq j : (\forall i : 0 \leq i < j : f \cdot U \cdot i \in U\,)\,)\,) \\
\equiv & \quad \{ \text{ property of } \lceil \ \} \\
 & (\forall U :: (\forall j : 0 \leq j : (\forall i : 0 \leq i < j : (f \cdot U \lceil j) \cdot i \in U\,)\,)\,) \\
\equiv & \quad \{ \text{ definition of } [\![\cdot]\!] \text{ (for finite lists) } \} \\
 & (\forall U :: (\forall j : 0 \leq j : [\![\, f \cdot U \lceil j \,]\!] \subseteq U\,)\,) \\
\equiv & \quad \{ \text{ swapping quantifications } \} \\
 & (\forall j : 0 \leq j : (\forall U :: [\![\, f \cdot U \lceil j \,]\!] \subseteq U\,)\,) \quad .
\end{array}
$$

This calculation shows that P_0 has the shape required by lemma 13 from the previous chapter – define $Q_j \cdot f$ as $(\forall U :: [\![\, f \cdot U \lceil j \,]\!] \subseteq U\,)$ –, from which we conclude that P_0 is uniformly admissible. Hence, what remains is to show that P_0 satisfies the premisses of theorem 9 from the previous chapter.

The first premiss, namely $(\exists f :: P_0 \cdot f)$, is easily satisfied, as this amounts to $(\exists f :: (\forall U :: [\![\, f \cdot U \,]\!] \subseteq U\,)\,)$, for which the function that maps any infinite set to the list consisting of infinitely many copies of its minimum is a valid instance.

To prove the second premiss, namely $(\forall f :: P_0 \cdot f \Rightarrow P_0 \cdot (F \cdot f)\,)$, we calculate as follows, for any function f:

$$
\begin{array}{ll}
 & P_0 \cdot (F \cdot f) \\
\equiv & \quad \{ \text{ definition (23) of } P_0 \ \} \\
 & (\forall U :: [\![\, F \cdot f \cdot U \,]\!] \subseteq U\,) \\
\equiv & \quad \{ \text{ definition (20) of } F \text{, where } b = \mathsf{min} \cdot U \ \} \\
 & (\forall U :: [\![\, b \triangleright f \cdot (U \backslash \{b\}\,)\,]\!] \subseteq U\,) \\
\equiv & \quad \{ \text{ property of } [\![\cdot]\!] \text{ and of } \subseteq \ \} \\
 & (\forall U :: b \in U \wedge [\![\, f \cdot (U \backslash \{b\}\,)\,]\!] \subseteq U\,) \\
\equiv & \quad \{ \ b = \mathsf{min} \cdot U \text{, hence } b \in U \ \} \\
 & (\forall U :: [\![\, f \cdot (U \backslash \{b\}\,)\,]\!] \subseteq U\,) \\
\Leftarrow & \quad \{ \ U \backslash \{b\} \subseteq U \text{ and transitivity of } \subseteq \ \} \\
 & (\forall U :: [\![\, f \cdot (U \backslash \{b\}\,)\,]\!] \subseteq U \backslash \{b\}\,) \\
\Leftarrow & \quad \{ \text{ instantiation } U := U \backslash \{b\} \ \} \\
 & (\forall U :: [\![\, f \cdot U \,]\!] \subseteq U\,) \\
\equiv & \quad \{ \text{ definition (23) of } P_0 \ \} \\
 & P_0 \cdot f \quad .
\end{array}
$$

This completes the proof of property (21).

* * *

We still have to prove properties (22) and (18), that is, $U \subseteq [\![\, lst \cdot U \,]\!]$ and $inc \cdot (lst \cdot U)$, for each (infinite) subset U of Nat.

The former of these two expresses that every element of set U occurs *somewhere* in $lst \cdot U$. Very likely, this proof will involve an indication of *where* in $lst \cdot U$ every element of U actually occurs.

We now begin proving property (22) by means of the following calculation:

$$
\begin{array}{ll}
 & U \subseteq [\![\, lst \cdot U \,]\!] \\
\equiv & \quad \{ \text{ property of } \subseteq \ \} \\
 & (\forall x : x \in U : x \in [\![\, lst \cdot U \,]\!]\,) \\
\equiv & \quad \{ \text{ property of } [\![\, \cdot \,]\!] \ \} \\
 & (\forall x : x \in U : (\exists i :: x = lst \cdot U \cdot i\,)\,) \\
\Leftarrow & \quad \{ \text{ instantiation, with function } h \text{ yet to be chosen } \} \\
 & (\forall x : x \in U : x = lst \cdot U \cdot (h \cdot U \cdot x)\,) \quad ,
\end{array}
$$

for all U. Because dummy i, in the above existential quantification, appears within the context of dummy x and free variable U, it generally depends on both x and U, hence the introduction of a function. For any infinite set U of naturals and for any element x of U, function application $h \cdot U \cdot x$ must be a natural number, namely the position of element x in list $lst \cdot U$.

Every natural number has finitely many predecessors: the predecessors of a natural number x are the natural numbers in the interval $[0\,..\,x)$, which is finite. Consequently, any element x in any set U of natural numbers has finitely many predecessors in U too, because the intersection $[0\,..\,x) \cap U$ is finite as well. Hence, for any (finite or infinite) set U of natural numbers and for every natural number x, the expression:

$$(\#\, u : 0 \leq u < x : u \in U\,)$$

is a well-defined natural number. We now define our auxiliary function h as follows, for all U and x:

$$(24) \qquad h \cdot U \cdot x \;=\; (\#\, u : 0 \leq u < x : u \in U\,) \quad .$$

Obviously, from the above introduction it follows immediately that:

$$h{\cdot}U{\cdot}x \;\leq\; x \quad .$$

In addition function h has the following property:

property 0 : If $b = \mathsf{min}{\cdot}U$ then, for all x, $x{\in}U \wedge x{\neq}b$, we have:

$$\begin{array}{lcl} h{\cdot}U{\cdot}b & = & 0 \\ h{\cdot}U{\cdot}x & = & h{\cdot}(U{\setminus}\{b\}){\cdot}x + 1 \end{array}$$

Notice that this property implies that $x \neq b$ if (and only if) $h{\cdot}U{\cdot}x \neq 0$.

□

From the above derivation we have obtained as remaining proof obligation:

$$(\forall x : x{\in}U : x = \mathit{lst}{\cdot}U{\cdot}(h{\cdot}U{\cdot}x)\,) \quad \text{, for all } U \quad ,$$

which we now prove using function h as defined in (24). We start with a little rewriting:

$$\begin{array}{ll} & (\forall U :: (\forall x : x{\in}U : x = \mathit{lst}{\cdot}U{\cdot}(h{\cdot}U{\cdot}x)\,)\,) \\ \equiv & \quad \{ \text{ 1-point rule, to introduce an additional dummy } \} \\ & (\forall U :: (\forall x : x{\in}U : (\forall j : h{\cdot}U{\cdot}x = j : x = \mathit{lst}{\cdot}U{\cdot}j\,)\,)\,) \\ \equiv & \quad \{ \text{ interchanging dummies } \} \\ & (\forall j :: (\forall U, x : x{\in}U \wedge h{\cdot}U{\cdot}x = j : x = \mathit{lst}{\cdot}U{\cdot}j\,)\,) \quad , \end{array}$$

and the formula thus obtained lends itself for proof by Mathematical Induction, namely over j; we start with case 0, and for all U and natural x, $x{\in}U$, we calculate:

$$\begin{array}{ll} & x \;=\; \mathit{lst}{\cdot}U{\cdot}0 \\ \equiv & \quad \{ \text{ definition (15), of } \mathit{lst} \text{ } \} \\ & x \;=\; \mathsf{min}{\cdot}U \\ \equiv & \quad \{ \; x{\in}U \text{, hence } \mathsf{min}{\cdot}U \leq x \; \} \\ & x \leq \mathsf{min}{\cdot}U \\ \equiv & \quad \{ \text{ definition of } \mathsf{min} \text{ } \} \\ & (\forall u : u{\in}U : x{\leq}u\,) \\ \equiv & \quad \{ \text{ predicate calculus } \} \end{array}$$

$$
\begin{array}{ll}
 & \neg(\exists u : 0 \le u < x : u \in U) \\
\equiv & \quad \{ \text{ property of } \# \ \} \\
 & (\# u : 0 \le u < x : u \in U) \ = \ 0 \\
\equiv & \quad \{ \text{ definition (24), of } h \ \} \\
 & h \cdot U \cdot x \ = \ 0 \quad .
\end{array}
$$

For the case of the positive naturals, that is the numbers of the shape $j+1$, for all natural j, we derive, for all U and natural x, $x \in U$:

$$
\begin{array}{ll}
 & x \ = \ lst \cdot U \cdot (j+1) \\
\equiv & \quad \{ \text{ definition (15), of } lst \text{, with } b = \mathsf{min} \cdot U \text{; property of } \triangleright \ \} \\
 & x \ = \ lst \cdot (U \backslash \{b\}) \cdot j \\
\Leftarrow & \quad \{ \text{ Induction Hypothesis, with } U := U \backslash \{b\} \ \} \\
 & x \in (U \backslash \{b\}) \ \wedge \ h \cdot (U \backslash \{b\}) \cdot x \ = \ j \\
\equiv & \quad \{ \text{ property of } \backslash \ \} \\
 & x \in U \ \wedge \ x \ne b \ \wedge \ h \cdot (U \backslash \{b\}) \cdot x \ = \ j \\
\equiv & \quad \{ \text{ property } 0 \ \} \\
 & x \in U \ \wedge \ x \ne b \ \wedge \ h \cdot U \cdot x \ = \ j+1 \\
\equiv & \quad \{ \text{ property } 0 \text{, using } h \cdot U \cdot x \ne 0 \ \} \\
 & x \in U \ \wedge \ h \cdot U \cdot x \ = \ j+1 \quad .
\end{array}
$$

This concludes the proof of property (22).

* * *

Finally, we are left with the obligation to prove that $lst \cdot U$ is increasing, for all infinite sets U of natural numbers. Predicate P_1, defined by:

$$
(25) \qquad P_1 \cdot f \ \equiv \ (\forall U :: inc \cdot (f \cdot U)) \quad ,
$$

is uniformly admissible. Hence, we can apply theorem 14 once more and conclude $(\forall U :: inc \cdot (lst \cdot U))$, provided we prove that P_1 satisfies the conditions of the theorem. In particular, we must show: $(\forall f :: P_1 \cdot f \Rightarrow P_1 \cdot (F \cdot f))$, and we do so by means of the following calculation:

$$
\begin{array}{ll}
 & P_1 \cdot (F \cdot f) \\
\equiv & \quad \{ \text{ definition (25) of } P_1 \ \} \\
 & (\forall U :: inc \cdot ((F \cdot f) \cdot U)) \quad ,
\end{array}
$$

and, for brevity's sake, we continue this calculation with the quantification's term in isolation, for all U:

$$
\begin{array}{ll}
 & inc\cdot((F\cdot f)\cdot U) \\
\equiv & \quad \{ \text{ definition (20) of } F \text{, where } b = \mathsf{min}\cdot U \ \} \\
 & inc\cdot(b \triangleright f\cdot(U\backslash\{b\})) \\
\equiv & \quad \{ \text{ property (12) of } inc \ \} \\
 & (\forall c : c\in[\![f\cdot(U\backslash\{b\})]\!] : b<c) \ \wedge \ inc\cdot(f\cdot(U\backslash\{b\})) \\
\Leftarrow & \quad \{ \text{ strengthening, by adding a conjunct } \} \\
 & [\![f\cdot(U\backslash\{b\})]\!] \subseteq U\backslash\{b\} \ \wedge \ (\forall c : c\in[\![f\cdot(U\backslash\{b\})]\!] : b<c) \ \wedge \\
 & inc\cdot(f\cdot(U\backslash\{b\})) \\
\Leftarrow & \quad \{ \text{ weakening the range, in the context of the new conjunct } \} \\
 & [\![f\cdot(U\backslash\{b\})]\!] \subseteq U\backslash\{b\} \ \wedge \ (\forall c : c\in U\backslash\{b\} : b<c) \ \wedge \\
 & inc\cdot(f\cdot(U\backslash\{b\})) \\
\equiv & \quad \{ \text{ definition of } \mathsf{min} \text{, using } b\in U \ \} \\
 & [\![f\cdot(U\backslash\{b\})]\!] \subseteq U\backslash\{b\} \ \wedge \ b = \mathsf{min}\cdot U \ \wedge \ inc\cdot(f\cdot(U\backslash\{b\})) \\
\equiv & \quad \{ \ b = \mathsf{min}\cdot U \ \} \\
 & [\![f\cdot(U\backslash\{b\})]\!] \subseteq U\backslash\{b\} \ \wedge \ inc\cdot(f\cdot(U\backslash\{b\})) \\
\Leftarrow & \quad \{ \text{ instantiation } U := U\backslash\{b\} \text{ (twice) } \} \\
 & (\forall U :: [\![f\cdot U]\!] \subseteq U) \ \wedge \ (\forall U :: inc\cdot(f\cdot U)) \\
\equiv & \quad \{ \text{ definitions (23) of } P_0 \text{, and (25) of } P_1 \ \} \\
 & P_0\cdot f \ \wedge \ P_1\cdot f \quad .
\end{array}
$$

So, this proof involves P_0 as well as P_1. Thus, we have not proved, as required, $(\forall f :: P_1\cdot f \Rightarrow P_1\cdot(F\cdot f))$, but instead we have proved the weaker $(\forall f :: P_0\cdot f \wedge P_1\cdot f \Rightarrow P_1\cdot(F\cdot f))$. By combining this with what we have established earlier, namely $(\forall f :: P_0\cdot f \Rightarrow P_0\cdot(F\cdot f))$, we obtain:

$$(\forall f :: P_0\cdot f \wedge P_1\cdot f \Rightarrow P_1\cdot(F\cdot f) \wedge P_1\cdot(F\cdot f)) \quad .$$

As a result the combined predicate $P_0 \wedge P_1$ satisfies the conditions of theorem 14 as well, and it also is uniformly admissible; hence, the combined conclusion $P_0\cdot lst \wedge P_1\cdot lst$ now follows from the theorem in one fell swoop.

aside: In the case of proofs by Mathematical Induction, combining two such proofs into one, so as to allow for mutual dependencies, gives rise

to what is known as a proof by "Simultaneous Induction". Note that, in a way, the premisses of the Productivity Theorems resemble the proof obligations of proofs by Mathematical Induction very much.

□

11.4.3 a simple example

As a very simple example we consider the list representation of the set of all natural powers of 2, that is, the set $(\mathsf{set}\, i : 0 \leq i : 2^i)$. This example is very simple indeed, because the problem can also be considered as a case of function listification, as discussed in Section 11.1. Here, however, we solve the problem by applying the technique for set listification.

The minimal element of the set $(\mathsf{set}\, i : 0 \leq i : 2^i)$ is 2^0, of course, and the set remaining after removal of this minimal element is $(\mathsf{set}\, i : 1 \leq i : 2^i)$. The expressions for these two sets have in common that they have the more general shape $(\mathsf{set}\, i : n \leq i : 2^i)$, for some natural n.

Thus generalizing the problem, we become interested in listification of $(\mathsf{set}\, i : n \leq i : 2^i)$, as a function of the natural parameter n. Calling this function f we obtain as its specification, for all natural n:

$$f \cdot n \;=\; lst \cdot (\mathsf{set}\, i : n \leq i : 2^i) \quad .$$

aside: Actually, we are dealing with a representation issue here: both the original set itself and the sets obtained from it after repeated removals of the minimal elements are represented by a single natural number: natural number n thus represents set $(\mathsf{set}\, i : n \leq i : 2^i)$.

□

For any natural n, the minimal element of $(\mathsf{set}\, i : n \leq i : 2^i)$ is 2^n, and the set remaining after removal of this minimal element is $(\mathsf{set}\, i : n{+}1 \leq i : 2^i)$. By substituting this into recursive definition (15), for function lst, we obtain the following (recursive) declaration for f:

$$f \cdot n \;=\; 2^n \triangleright f \cdot (n{+}1) \quad .$$

If so desired, subexpression 2^n can be eliminated by means of the standard technique of an additional parameter, representing the value of this subexpression. This leads to the introduction of a new function g, with an additional parameter and with this specification, for all x and natural n:

$$g \cdot x \cdot n \;=\; f \cdot n \quad \text{, provided } x = 2^n \quad .$$

Then we have – after all, $2^0 = 1$ – :

$$f \cdot 0 \;=\; g \cdot 1 \cdot 0 \quad ,$$

and, hence:

$$lst \cdot (\mathsf{set}\, i : 0 \le i : 2^i\,) \;=\; g \cdot 1 \cdot 0 \quad .$$

As (recursive) declaration for g we then obtain:

$$g \cdot x \cdot n \;=\; x \triangleright g \cdot (2 * x) \cdot (n+1) \quad .$$

11.4.4 merging infinite lists

The union of two infinite sets is infinite too. Therefore, it stands to reason to ask for a function $\mathbf{m}$ – "merge" – that maps two increasing infinite lists x, y of naturals to the increasing infinite list of naturals representing the union of the sets represented by x and y. (Because set union is associative so is $\mathbf{m}$, which is why we write it as a binary operator in infix notation.) That is, $\mathbf{m}$ is specified by, for all x, y in $\mathcal{L}_\infty(\mathsf{Nat})$:

$$(26) \qquad inc \cdot x \;\wedge\; inc \cdot y \;\Rightarrow\; inc \cdot (x \,\mathbf{m}\, y) \;\wedge\; [\![\, x \,\mathbf{m}\, y \,]\!] \;=\; [\![\, x \,]\!] \cup [\![\, y \,]\!] \quad .$$

Now it is not very difficult – we leave this as an exercise – to derive the following, recursive and productive, declaration for $\mathbf{m}$; here operator $\mathbf{m}$ is supposed to have higher binding power than $\triangleright$:

$$(27) \qquad (b \triangleright x)\; \mathbf{m}\; (c \triangleright y) \;=\; \begin{array}{lll} \mathsf{if}\; b < c & \rightarrow & b \triangleright x \,\mathbf{m}\, (c \triangleright y) \\ [\!]\; b = c & \rightarrow & b \triangleright x \,\mathbf{m}\, y \\ [\!]\; c < b & \rightarrow & c \triangleright (b \triangleright x) \,\mathbf{m}\, y \\ \mathsf{fi} & & \end{array}$$

11.5 Recursively Defined Sets

11.5.0 a simple example

The set $(\mathsf{set}\, i : 0 \le i : 2^i\,)$ can also be defined *recursively*, in the following way. Given that $2^0 = 1$ and that $2^{i+1} = 2 * 2^i$, it is the *smallest* of all sets U that satisfy both:

$$1 \in U \quad \text{, and:}$$

$$(\forall x : x \in U : 2 * x \in U\,) \quad .$$

The latter of these two requirements can be rendered more concisely, by means of the "map" operator on sets, thus:

$$(2*) \bullet U \subseteq U \quad ,$$

whereas the former can be rewritten as $\{1\} \subseteq U$. By combining these two we can redefine our set, equivalently, as the smallest of all sets U satisfying:

$$(28) \qquad \{1\} \cup (2*) \bullet U \ \subseteq \ U \quad .$$

By Knaster-Tarski's theorem [exercise 0, in Section 11.7] – after all, functions $(\{1\} \cup)$ and $((2*) \bullet)$ both are monotonic with respect to $\subseteq$, and so is their composition – this smallest set is equal to the smallest of all sets U satisfying:

$$(29) \qquad \{1\} \cup (2*) \bullet U \ = \ U \quad .$$

The increasing infinite list representing this set can now be specified as an infinite list s of which the set represented by it solves (29), and that is increasing, thus:

$$(30) \qquad [\![s]\!] \ = \ \{1\} \cup (2*) \bullet [\![s]\!] \ \wedge \ inc \cdot s \quad .$$

If s is an infinite list and if s satisfies (30), then $[\![s]\!]$ indeed is *a* solution to (29). That it also is the *smallest* solution remains to be seen, but we postpone the discussion of this for a while. We now derive:

$$\begin{array}{ll}
 & [\![s]\!] \ = \ \{1\} \cup (2*) \bullet [\![s]\!] \ \wedge \ inc \cdot s \\
\equiv & \quad \{ \bullet \text{ and } [\![\cdot]\!] \text{ commute (11) } \} \\
 & [\![s]\!] \ = \ \{1\} \cup [\![(2*) \bullet s]\!] \ \wedge \ inc \cdot s \\
\equiv & \quad \{ [\![\cdot]\!] \text{ over } \triangleright \ \} \\
 & [\![s]\!] \ = \ [\![1 \triangleright (2*) \bullet s]\!] \ \wedge \ inc \cdot s \\
\equiv & \quad \{ \text{ to be shown below } \} \\
 & [\![s]\!] \ = \ [\![1 \triangleright (2*) \bullet s]\!] \ \wedge \ inc \cdot s \ \wedge \ inc \cdot (1 \triangleright (2*) \bullet s) \\
\equiv & \quad \{ \text{ lemma "uni" } \} \\
 & s \ = \ 1 \triangleright (2*) \bullet s \ \wedge \ inc \cdot s \ \wedge \ inc \cdot (1 \triangleright (2*) \bullet s) \\
\equiv & \quad \{ \text{ to be shown below too } \} \\
 & s \ = \ 1 \triangleright (2*) \bullet s \quad .
\end{array}$$

We now define s by $s = 1 \triangleright (2*) \bullet s$. This definition is productive; as a result value s, thus defined, is an infinite list and it is *unique*. Because all steps in this derivation are equivalence preserving, and because all infinite subsets of the naturals are representable as increasing infinite lists, we conclude that the solution to equation (29) is unique as well. Hence, thus defined list s represents the smallest solution of (29): obviously, the unique solution to an equation is its smallest solution.

In the above derivation we still have to fill two gaps, which happen to be fairly similar, though. Firstly, we have to show that:

(31) $\quad [\![s]\!] = [\![1 \triangleright (2*) \bullet s]\!] \ \wedge \ inc \cdot s \ \Rightarrow \ inc \cdot (1 \triangleright (2*) \bullet s)$, for all s ,

and, secondly, we must prove that:

(32) $\quad s = 1 \triangleright (2*) \bullet s \ \Rightarrow \ inc \cdot s \ \wedge \ inc \cdot (1 \triangleright (2*) \bullet s)$, for all s .

Proof obligation (31) can be safely strengthened to:

(33) $\quad inc \cdot s \ \Rightarrow \ inc \cdot (1 \triangleright (2*) \bullet s)$, for all natural infinite lists s .

To prove this we derive:

$$
\begin{array}{ll}
 & inc \cdot (1 \triangleright (2*) \bullet s) \\
\equiv & \quad \{ \text{ property (13) of } inc \ \} \\
 & 1 < ((2*) \bullet s) \cdot 0 \ \wedge \ inc \cdot ((2*) \bullet s) \\
\Leftarrow & \quad \{ \text{ properties of } \bullet \ \} \\
 & 1 < 2*(s \cdot 0) \ \wedge \ inc \cdot (2*) \ \wedge \ inc \cdot s \\
\Leftarrow & \quad \{ \text{ strenghtening; } (2*) \text{ is increasing } \} \\
 & 1 \leq s \cdot 0 \ \wedge \ inc \cdot s \ ,
\end{array}
$$

and here the calculation comes to a grinding halt because we cannot conclude the first conjunct, $1 \leq s \cdot 0$, from anything else but $s = 1 \triangleright (2*) \bullet s$.

To save this we, therefore, define a predicate P as follows, for all infinite lists s:

(34) $\quad P \cdot s \ \equiv \ 1 \leq s \cdot 0 \ \wedge \ inc \cdot s$.

Predicate P is admissible and now it is a straightforward exercise to prove that it satisfies:

$$P\cdot s \;\Rightarrow\; P\cdot(1 \triangleright (2*)\bullet s) \quad \text{, for all natural infinite lists } s \quad ,$$

as an alternative to property (33).

Thus, predicate P satisfies the conditions of the Second Productivity Theorem, which allows us to conclude:

(35) $\quad s \;=\; 1 \triangleright (2*)\bullet s \;\Rightarrow\; P\cdot s \quad$, for all s ,

which also proves (32), and this concludes the solution of this problem.

11.5.1 a few more general patterns

The recursive definition of set $(\mathsf{set}\, i : 0 \le i : 2^i)$ is a specific instance of a set defined in the following way. For given $b : b \in \mathsf{Nat}$ and for f a given function of type $\mathsf{Nat} \to \mathsf{Nat}$ set V is the *smallest* of all sets U satisfying:

(36) $\quad b \in U \;\wedge\; (\forall n :: n \in U \Rightarrow f\cdot n \in U)$,

which, as before, can also be written as:

(37) $\quad \{b\} \cup f\bullet U \;\subseteq\; U$.

As before, by Knaster Tarski's theorem – again, functions $(\{b\}\cup)$ and $(f\bullet)$ both are monotonic – this smallest set is the same as the smallest of all sets U satisfying:

(38) $\quad \{b\} \cup f\bullet U \;=\; U$.

An infinite increasing list representing this set now is an infinite list s for which $[\![s]\!]$ is a solution of (38). That it also is the smallest solution remains to be seen; as before, we will show this by showing that the solution is unique. Again, that $[\![s]\!]$ solves (38) is rendered by this specification:

$$[\![s]\!] \;=\; \{b\} \cup f\bullet [\![s]\!] \;\wedge\; inc\cdot s \quad .$$

The derivation of a recursive definition for s is essentially the same as in the previous case. After all, in the previous derivation almost nowhere we have used particular properties of constant 1 or of function $(2*)$; the only exception occurs in the proof that the list is increasing, so this part must be redone.

In the previous, special case, the essential property turned out to be:

$$P\cdot s \;\Rightarrow\; P\cdot(1 \triangleright (2*)\bullet s) \quad \text{, for all natural infinite lists } s \quad ,$$

where P was defined by:

(34) $\quad P\cdot s \;\equiv\; 1 \leq s\cdot 0 \;\wedge\; inc\cdot s \quad .$

For the current, more general, problem we have to define P as follows:

$$P\cdot s \;\equiv\; b \leq s\cdot 0 \;\wedge\; inc\cdot s \quad ,$$

and then the proof is as straightforward as in the previous case. Thus, we obtain as recursive definition for s:

$$s \;=\; b \triangleright f\bullet s \quad .$$

* * *

An even more general pattern arises if function f is replaced by any monotonic function φ, mapping infinite subsets of Nat to infinite subsets of Nat. In this case we define set V as the smallest of all sets U satisfying:

$$\{b\} \cup \varphi\cdot U \;\subseteq\; U \quad .$$

Again, by monotonicity of $(\{b\}\cup)$ and φ we conclude, again using Knaster-Tarski, that V also is the smallest of all sets U satisfying:

(39) $\quad U \;=\; \{b\} \cup \varphi\cdot U \quad .$

Formula (39) determines V uniquely; we shall prove this as a corollary of the derivation to follow.

An increasing infinite list x representing V is an increasing infinite list that solves specification (39); so, such infinite list x can be specified by:

(40) $\quad [\![x]\!] = \{b\} \cup \varphi\cdot[\![x]\!] \;\wedge\; inc\cdot x \quad .$

Because every infinite subset of Nat is representable as an infinite list, a one-to-one correspondence exists between the solutions of (39) and (40). Hence, if the solution of (40) is unique, then so is the solution of (39). Therefore, to prove that (39) has a unique solution it suffices to prove that (40) has a unique solution.

In its full generality, this problem is hard to solve, however. The general strategy for enumerating a subset of the naturals requires we be able to identify, among others, the minimal element of set V, but for arbitrary functions

φ this may be impossible. After all, for some, possibly very large, argument function φ may have a value *smaller than* b and, therefore, we cannot simply assume that b is V's least element.

To avoid this difficulty we assume that φ has the following additional property:

(41) $\quad (\forall c : c \in U : b \leq c) \;\Rightarrow\; (\forall c : c \in \varphi \cdot U : b < c)$, for all U .

To be able to obtain a useful recursive declaration for x we need an implementation, in the realm of increasing infinite lists of naturals, of function φ . That is, as part of the solution of the problem, we need (to construct a declaration for) a function F , say, of type $\mathcal{L}_\infty(\mathsf{Nat}) \to \mathcal{L}_\infty(\mathsf{Nat})$, and with this specification, for all x in $\mathcal{L}_\infty(\mathsf{Nat})$ – notice that this specification just expresses that F implements φ –:

(42) $\quad inc \cdot x \;\Rightarrow\; [\![F \cdot x]\!] = \varphi \cdot [\![x]\!]$.

(43) $\quad inc \cdot x \;\Rightarrow\; inc \cdot (F \cdot x)$.

In what follows we assume this is possible; whether or not this is true in any particular instance of this problem, depends, of course, on the properties of function φ .

Now we derive:

$$
\begin{array}{ll}
 & [\![x]\!] = \{b\} \cup \varphi \cdot [\![x]\!] \;\wedge\; inc \cdot x \\
\equiv & \quad \{ \text{ specification (42) of } F \text{ } \} \\
 & [\![x]\!] = \{b\} \cup [\![F \cdot x]\!] \;\wedge\; inc \cdot x \\
\equiv & \quad \{ \text{ property of } [\![\cdot]\!] \text{ and } \triangleright \text{ } \} \\
 & [\![x]\!] = [\![b \triangleright F \cdot x]\!] \;\wedge\; inc \cdot x \\
\equiv & \quad \{ \; inc \cdot x \Rightarrow inc \cdot (b \triangleright F \cdot x) \text{ , see below } \} \\
 & [\![x]\!] = [\![b \triangleright F \cdot x]\!] \;\wedge\; inc \cdot x \;\wedge\; inc \cdot (b \triangleright F \cdot x) \\
\equiv & \quad \{ \text{ lemma “uni” } \} \\
 & x = b \triangleright F \cdot x \;\wedge\; inc \cdot x \;\wedge\; inc \cdot (b \triangleright F \cdot x) \\
\equiv & \quad \{ \text{ Leibniz } \} \\
 & x = b \triangleright F \cdot x \;\wedge\; inc \cdot x \\
\equiv & \quad \{ \text{ Productivity Theorem, for predicate } inc \text{ } \} \\
 & x = b \triangleright F \cdot x \quad .
\end{array}
$$

Both the appeal to lemma "uni" and the appeal to the Productivity Theorem requires $inc \cdot x \Rightarrow inc \cdot (b \triangleright F \cdot x)$, for all infinite lists x of naturals. To prove this, we calculate:

$$
\begin{array}{ll}
 & inc \cdot (b \triangleright F \cdot x) \\
\equiv & \quad \{ \text{ property (12) of } inc \ \} \\
 & (\forall c : c \in [\![F \cdot x]\!] : b < c) \ \wedge \ inc \cdot (F \cdot x) \\
\Leftarrow & \quad \{ \text{ (42) and (43) } \} \\
 & (\forall c : c \in \varphi \cdot [\![x]\!] : b < c) \ \wedge \ inc \cdot x \\
\Leftarrow & \quad \{ \text{ property (41) of } \varphi \text{ , with } U := [\![x]\!] \ \} \\
 & (\forall c : c \in [\![x]\!] : b \leq c) \ \wedge \ inc \cdot x \\
\equiv & \quad \{ \ \forall / \mathsf{min}\text{-connection} \ \} \\
 & b \leq (\mathsf{min}\, c : c \in [\![x]\!] : c) \ \wedge \ inc \cdot x \\
\equiv & \quad \{ \ inc \cdot x \ \} \\
 & b \leq x \cdot 0 \ \wedge \ inc \cdot x \quad ,
\end{array}
$$

and, as before, here the derivation comes to a halt again. So, now we need a predicate P defined by:

$$P \cdot x \ \equiv \ b \leq x \cdot 0 \ \wedge \ inc \cdot x \quad ,$$

which is admissible and which does the job.

11.5.2 Hamming's exercise

We consider a set V of natural numbers, defined recursively as (the smallest set satisfying):

$$
\begin{array}{l}
1 \in V \quad \text{, and:} \\
2 * b \in V \ \wedge \ 3 * b \in V \ \wedge \ 5 * b \in V \quad \text{, for all } b \in V \quad .
\end{array}
$$

In words, V is the set of all natural numbers having no other prime factors than $2, 3, 5$. The problem now is to construct a productive declaration for the increasing infinite list representing V.

This problem is an instance of the last abstract pattern discussed in the previous subsection. We define constant b and function φ as follows, for all infinite subsets U of Nat:

$$\begin{array}{rcl} b & = & 1 \\ \varphi\cdot U & = & (2*)\bullet U \cup (3*)\bullet U \cup (5*)\bullet U \end{array}$$

This particular choice of b and φ satisfies requirement (41), which was needed for the solution of the general case:

(41) $\quad (\forall c : c\in U : b\leq c) \;\Rightarrow\; (\forall c : c\in\varphi\cdot U : b<c)$, for all U .

The solution to the current problem now is an instance of the solution to the general pattern; all we have to do now is to construct an implementation of φ in terms of infinite lists. This, however, is fairly easy: $\bullet$ on sets is implemented by $\bullet$ on lists, and $\cup$ is implemented by operator $\mathbf{m}$ on lists, as discussed in subsection 11.4.4. Thus, function φ is implemented by function F defined by:

$$F\cdot x \;=\; 2\bullet x \;\mathbf{m}\; 3\bullet x \;\mathbf{m}\; 5\bullet x \quad .$$

As a result, a declaration for the list representing Hamming's set is:

$$x \;=\; 1 \;\triangleright\; 2\bullet x \;\mathbf{m}\; 3\bullet x \;\mathbf{m}\; 5\bullet x \quad .$$

11.6 Transposing an Infinite Matrix

The elements of a list may be lists as well. The type $\mathcal{L}_\infty(\mathcal{L}_\infty)$ is the set of all infinite lists with infinite lists as their elements. If, for some type B, $x\in\mathcal{L}_\infty(\mathcal{L}_\infty(B))$ then $x\cdot j\in\mathcal{L}_\infty(B)$ and $x\cdot j\cdot i\in B$, for all natural j,i. Now we can specify functions, of type $\mathcal{L}_\infty(\mathcal{L}_\infty)\to\mathcal{L}_\infty(\mathcal{L}_\infty)$, such as function trp with specificiation, for all x in $\mathcal{L}_\infty(\mathcal{L}_\infty)$:

$$(\forall j,i :: trp\cdot x\cdot j\cdot i = x\cdot i\cdot j\,) \quad .$$

We may view $x\in\mathcal{L}_\infty(\mathcal{L}_\infty(B))$ as an infinite matrix with elements of type B; in this view $trp\cdot x$ then is the transposition of infinite matrix x.

We derive a declaration for trp , as follows; as usual, we keep in mind that we are heading for a list (of lists) for the function's value:

$$\begin{array}{ll} & trp\cdot x\cdot j\cdot i \\ = & \quad \{\text{ specification of } trp \text{ }\} \\ & x\cdot i\cdot j \\ = & \quad \{\text{ sectioning, so as to isolate } i \text{ }\} \end{array}$$

$$
\begin{array}{ll}
 & (\cdot j)\cdot(x\cdot i) \\
= & \quad \{\ x \text{ is a list: property of } \bullet\ \} \\
 & ((\cdot j)\bullet x)\cdot i \quad ,
\end{array}
$$

from which we conclude that *trp*'s specification is satisfied if we are able to establish, for all x:

(44) $\quad (\forall j :: \mathit{trp}\cdot x\cdot j = (\cdot j)\bullet x) \quad .$

For every list x and natural j the expression $(\cdot j)\bullet x$ is a list, but the function mapping j to this expression is not a list yet. To listify it we derive, with induction on j:

$$
\begin{array}{ll}
 & \mathit{trp}\cdot x\cdot 0 \\
= & \quad \{\ (44), \text{ as new specification of } \mathit{trp}\ \} \\
 & (\cdot 0)\bullet x \\
= & \quad \{\ \triangleright\text{-trick}\ \} \\
 & (\,(\cdot 0)\bullet x \ \triangleright\ ?\,)\cdot 0
\end{array}
$$

and:

$$
\begin{array}{ll}
 & \mathit{trp}\cdot x\cdot(j{+}1) \\
= & \quad \{\ (44)\ \} \\
 & (\cdot(j{+}1))\bullet x \\
= & \quad \{\ (\cdot(j{+}1)) \ = \ (\cdot j)\circ(\lfloor 1)\ \} \\
 & ((\cdot j)\circ(\lfloor 1))\bullet x \\
= & \quad \{\ \text{connection of } \circ \text{ and } \bullet\ \} \\
 & (\cdot j)\bullet((\lfloor 1)\bullet x) \\
= & \quad \{\ (44), \text{ by Induction Hypothesis}\ \} \\
 & \mathit{trp}\cdot((\lfloor 1)\bullet x)\cdot j \\
= & \quad \{\ \triangleright\text{-trick}\ \} \\
 & (\,?\ \triangleright\ \mathit{trp}\cdot((\lfloor 1)\bullet x)\,)\cdot(j{+}1) \quad .
\end{array}
$$

Thus we obtain as declaration for *trp*:

$$\mathit{trp}\cdot x \ = \ (\cdot 0)\bullet x \ \triangleright\ \mathit{trp}\cdot((\lfloor 1)\bullet x) \quad ,$$

which can also be written as:

$$\begin{array}{lcl} trp\cdot x & = & t \triangleright trp\cdot y \\ & & \mathsf{whr}\ \ t = (\cdot 0)\bullet x \ \ \&\ \ y = (\lfloor 1)\bullet x\ \ \mathsf{end} \end{array}$$

The expressions $(\cdot 0)\bullet x$ and $(\lfloor 1)\bullet x$ have similar recursive properties. By first substituting $s\triangleright x$ for x and then substituting $b\triangleright s$ for s we obtain, after a few simplifications that involve properties of $\bullet$ and $\triangleright$ only:

$$(\cdot 0)\bullet((b\triangleright s)\triangleright x) \ = \ b \triangleright (\cdot 0)\bullet x \ \ .$$

Similarly, we obtain:

$$(\lfloor 1)\bullet((b\triangleright s)\triangleright x) \ = \ s \triangleright (\lfloor 1)\bullet x \ \ .$$

By means of tupling we combine these patterns in a single function h, with specification, for all x:

$$h\cdot x \ = \ \langle\, (\cdot 0)\bullet x\,,\, (\lfloor 1)\bullet x\,\rangle \ \ .$$

A recursive declaration for h then is:

$$h\cdot((b\triangleright s)\triangleright x) \ = \ \langle\, b\triangleright t\,,\, s\triangleright y\,\rangle \ \ \mathsf{whr}\ \langle t, y\rangle = h\cdot x\ \mathsf{end} \ \ ,$$

and using this h we rewrite the declaration for trp into:

$$trp\cdot x \ = \ t \triangleright trp\cdot y \ \ \mathsf{whr}\ \langle t, y\rangle = h\cdot x\ \mathsf{end} \ \ .$$

11.7 Exercises

0. We consider a set U with a *partial order* relation $\sqsubseteq$ ("under"). A function f of type $U\to U$ is called *monotonic* (with respect to $\sqsubseteq$) if it satisfies:

$$(\forall u, v : u, v\in U : u\sqsubseteq v \ \Rightarrow\ f\cdot u\sqsubseteq f\cdot v\,) \ \ .$$

For any predicate R on U, a value x in U is called a *least solution* of the equation $u : R\cdot u$ if and only if both $R\cdot x$ $-x$ is a solution$-$ and $(\forall u :: R\cdot u \Rightarrow x\sqsubseteq u\,)$ $-x$ is under all solutions$-$.

(a) Under the assumption that, for a given predicate R, the equation $u : R\cdot u$ has a least solution, prove that it is unique.

(b) (Knaster-Tarski's theorem.) We consider a monotonic function f in $U \to U$. Prove that the least solutions, given that they exist, of the following two equations are equal:

$$u : f \cdot u \sqsubseteq u \quad \text{and:} \quad u : f \cdot u = u \quad .$$

1. Function $fusc$, of type $\mathsf{Nat} \to \mathsf{Nat}$, is given by, for all natural n:

$$\begin{array}{lcl} fusc \cdot 1 & = & 1 \\ fusc \cdot (2 * n) & = & fusc \cdot n \\ fusc \cdot (2 * n + 1) & = & fusc \cdot (n+1) \; + \; fusc \cdot n \end{array}$$

   (a) Prove that this definition implies $fusc \cdot 0 \; = \; 0$.

   (b) Derive a declaration for an increasing infinite list representing $fusc$.

2. Derive a declaration for a function f, having type $\mathcal{L}_\infty(\mathsf{Int}) \to \mathcal{L}_\infty(\mathsf{Int})$, and specified by:

$$(\forall s, j :: f \cdot s \cdot j = rev \cdot (s \lceil j)) \quad ,$$

3. Let $\mathsf{Nat}+$ be the set $\mathsf{Nat} \cup \{\infty\}$, where ∞ is an additional value with no other property than that $(\forall i : i \in \mathsf{Nat} : i < \infty)$. As in Section 11.4 we consider function lst, but now for all infinite subsets of $\mathsf{Nat}+$, and as before defined by:

$$lst \cdot U \;\; = \;\; b \triangleright lst \cdot (U \backslash \{b\}) \;\; \mathsf{whr}\; b \; = \; \mathsf{min} \cdot U \;\; \mathsf{end} \quad .$$

   (a) Show that $[\![lst \cdot U]\!] \subseteq U$, for all $U \subseteq \mathsf{Nat}+$.

   (b) Show that $[\![lst \cdot U]\!] \neq U$, if $\infty \in U$.

   (c) Apparently, the correctness argument given in Section 11.4 fails here, because, apparently, Nat has a property that $\mathsf{Nat}+$ lacks. Identify this property.

4. Derive a declaration for an increasing infinite list representing the set of all natural numbers of the shape $2^m + 3^n$, for all natural m, n.

5. Derive a declaration for a function f, having type $\mathcal{L}_\infty(\mathsf{Int}) \to \mathcal{L}_\infty(\mathsf{Int}) \to \mathcal{L}_\infty(\mathsf{Int})$, and specified by, for all infinite integer lists x, y:

$$\begin{array}{l} inc\cdot x \ \wedge \ inc\cdot y \ \Rightarrow \\ inc\cdot(f\cdot x\cdot y) \ \wedge \ [\![f\cdot x\cdot y]\!] \ = \ (\mathsf{set}\, i,j : 0\leq i \wedge 0\leq j : x\cdot i+y\cdot j\,) \quad . \end{array}$$

6. Prove lemma "uni"; that is, prove, for infinite integer lists x, y:

$$inc\cdot x \ \wedge \ inc\cdot y \ \Rightarrow \ (\,[\![x]\!] = [\![y]\!] \ \equiv \ x = y\,) \quad .$$

7. (a) Derive declaration (27), for $\mathbf{m}$ –see Section 11.4.4–, from specification (26).
   (b) Prove that $\mathbf{m}$ is associative.

8. Specify and derive a recursive declaration for a function mapping two increasing infinite lists of naturals x, y to the increasing infinite list of naturals representing the *intersection* of the sets represented by x and y. Under which additional condition is this function's value really an infinite list? What can be said about the function's value if this condition is not met?

9. We consider function *zip*, of type $\mathcal{L}_\infty(B) \to \mathcal{L}_\infty(B) \to \mathcal{L}_\infty(B)$, for some element type B, according to this (element-wise) specification, for all infinite lists x, y and for all natural i:

$$zip\cdot x\cdot y\cdot(2{*}i) \ = \ x\cdot i \quad \text{, and: } \ zip\cdot x\cdot y\cdot(2{*}i{+}1) \ = \ y\cdot i \quad .$$

   This specification tells us that infinite list $zip\cdot x\cdot y$ is obtained from x and y by interleaving the elements of x and y strictly alternatingly: the elements of x occur in $zip\cdot x\cdot y$ at the even positions whereas y's elements occur at the odd positions.

   As function *zip* destroys no information, it has an inverse, consisting of a pair of functions *unze* ("unzip even") and *unzo* ("unzip odd"), say, mapping infinite lists to infinite lists and with these specifications, for all z and for all natural i:

$$\begin{array}{lcl} unze\cdot z\cdot i & = & z\cdot(2{*}i) \\ unzo\cdot z\cdot i & = & z\cdot(2{*}i{+}1) \end{array}$$

   (a) Using the above specifications of *zip*, *unze*, and *unzo*, prove that *unze* and *unzo* together indeed constitute *zip*'s inverse; that is prove that they satisfy, for all infinite lists x, y:

$$unze\cdot(zip\cdot x\cdot y) \ = \ x \quad \text{, and: } \ unzo\cdot(zip\cdot x\cdot y) \ = \ y \quad .$$

(b) Derive recursive declarations for *zip* , *unze* , and *unzo* .

10. *the Thue-Morse sequence*: We consider a function pr , of type $\mathsf{Nat} \to \{0, 1\}$, mapping the natural numbers to their, so-called, parities. A recursive definition for pr is given as follows, for all natural n :

$$\begin{array}{lcl} pr \cdot 0 & = & 0 \\ pr \cdot (2 * n) & = & pr \cdot n \\ pr \cdot (2 * n + 1) & = & 1 - pr \cdot n \end{array}$$

Thus, $pr \cdot n = 0$ if and only if the number of ones in n 's binary representation is even, and otherwise $pr \cdot n = 1$. Derive a (recursive) declaration, in which pr does not occur, for an infinite list ms representing pr .

Bibliography

[1] R.S. Bird, *Introduction to functional programming.* Prentice Hall International, Hemel Hempstead, 1998.

[2] A.J.M. van Gasteren, *On the Shape of Mathematical Arguments*, LNCS445, Springer-Verlag, 1990.

[3] D. Gries, F.B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1994