Programming by Calculation

Rob R. Hoogerwoord
department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

June 7, 1996

0 Introduction

Functional programming lends itself very well for a calculational style of reasoning, in which programs are developed by means of largely *uninterpreted* formula manipulation, in very much the same way as a physicist solves a problem by means of the differential and integral calculus, without worrying about the physical interpretation of his intermediate formulae.

In the mathematical tradition formal reasoning is considered as laborious, tedious and, therefore, awkward, but this is largely caused by the use of inadequate formalisms, which were not designed for effective problem solving in the first place. To make calculational reasoning effective we must use a formalism that is simple and concise, and that is not burdened by a baroque syntax. As an example, although *type information* about the variables in a formula is important, we should not be forced to drag along type information in our calculations, because this information remains largely *constant* whereas in calculations we are only interested in opportunities for *changes*. This is the main reason why I prefer to carry out my calculations in a formalism that is, at least syntactically, essentially typeless.

The choice of the notation is, therefore, extremely important. Just as the Hindu-Arabic numerals are much better suited for performing multiplications and long divisions than are the Roman numerals, an algebraic predicate *calculus*, such as developed in [1], is better geared towards calculational reasoning than a deductive logic based on inference rules. The algebraic properties of the operators in a formalism form a main source of possibilities for formula manipulation. Therefore, the formalism must be such that these properties can be identified and made explicit in a clear way. The rigourous distinction, in a deductive logic, of several levels of formalism may be necessary for the purpose for which such a logic is devised, namely the development of metatheories *about* the formalism. For effective reasoning *in* the formalism, however, this very rigourous distinction becomes a major stumbling block, because, first, the same algebraic properties tend to emerge in multitude on the different levels, and,

second, additional rules are required to connect these levels. The result is an unduly complicated formalism. (See Section 0.1 for an example.)

Uninterpreted formula manipulation is only possible if the formalism is defined unambiguously and if the manipulation rules are stated explicitly: the symbols will be doing the work, and each step in a calculation must be justified by an appeal to these rules. As a result, calculational derivations can be verified, if so desired even mechanically, and it becomes impossible to use knowledge about the underlying model without making this knowledge explicit first. This is a definite advantage over the informal and implicit style of reasoning that is common practice in traditional mathematics.

This is not meant to say that calculational reasoning is a blind, mechanical process. It is not, and although it is a very goal-directed activity we do have to make, every now and then, *design decisions* about the direction in which our solution will evolve.

* * *

In this contribution I wish to show how a number of important techniques for functional programming are effectively supported by calculational reasoning. Using relatively simple (but non-trivial) examples we shall address the following topics:

- implicit specifications and their solutions
- generalisation by abstraction
- program transformations

The remainder of this introduction is devoted to the notation and the associated rules of the game, as needed in this paper. For a justification of these rules we refer to [2]. Moreover, we shall introduce the calculational style by means of a few simple examples.

0.0 function application

The only thing we can do with a *function* is *apply* it to an *argument*. For this purpose we use a binary operator $\cdot$ ("dot"), that is, we write the application of function f to argument x as $f \cdot x$, not as $f(x)$ (as in traditional mathematics) nor as $f\ x$ (as in some other functional programming languages). By using an explicit symbol we recognize that function application is a binary operation like any other one: everything that applies to binary operators in general applies to function application as well.

The one-and-only property of function application is Leibniz's rule of substitution of equals for equals:

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

In functional programming functions are treated on equal footing with other values; in particular, functions may be used as arguments in applications and the values of applications may be functions again. For example, even the application of a function to itself, as in $f \cdot f$, is permissible and sometimes meaningful.

The value $f \cdot x$ may be a function again; if it is it can be applied to yet another argument, as in $(f \cdot x) \cdot y$. To save parentheses we adopt the convention that $\cdot$ is *left-binding*, that is, we read $f \cdot x \cdot y$ as $(f \cdot x) \cdot y$. Such functions can be used as multi-argument functions: we consider an expression like $f \cdot x \cdot y$ as the application of a two-argument function f to arguments x and y.

0.1 function types

A *type* is a common property of a collection of values. Types play an imporant role in programming, but to understand the rules of functional programming, we do not need an elaborate syntactic theory of types; for our purpose, it suffices to view types as predicates on the value space of the functional language [2].

Types are particularly important in connection with functions: if for every x of type U the application $f \cdot x$ has type V then we record this by stating that f has type $U \rightarrow V$. So, the function type $U \rightarrow V$ is the common property of "mapping elements of U to elements of V".

Formally, $U \rightarrow V$ can be defined as a predicate, for any two predicates U and V, as follows, where we use lambda-notation, $\lambda x.E$, for the function mapping x to (the value of) expression E:

$$(U \rightarrow V) \cdot (\lambda x.E) \quad \equiv \quad (\forall x :: U \cdot x \Rightarrow V \cdot E) \quad .$$

In an inference rule based deduction system this equivalence must be rendered as two separate and quite different rules, the so-called introduction and elimination rules for the type constructor $\rightarrow$:

$$\frac{\Gamma, x : U \ \vdash \ E : V}{\Gamma \ \vdash \ (\lambda x.E) : U \rightarrow V} \qquad \frac{\Gamma \ \vdash \ (\lambda x.E) : U \rightarrow V \ , \ \Gamma \ \vdash \ F : U}{\Gamma \ \vdash \ E(x := F) : V}$$

Apart from the abundance of redundant Γs the mere shape of these two rules obfuscates that they are actually the two halves of a simple equivalence.

0.2 function definitions

A *function definition* has the shape

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

where E is an expression in which the names x and y may occur as free variables. This defines f as a two-argument function; x and y are the *parameters* of the function and E is called f's *defining expression*. In this case f has two parameters, but a function can have any number of parameters, even none, in which case it is called a *constant*. If f itself also occurs in its defining expression E, which is permitted, the definition is *recursive*.

Such a function definition is an abbreviation, as it contains an implicit universal quantification over the function's parameters:

$$(\forall x, y :: f \cdot x \cdot y = E) \quad ,$$

from which we obtain the following rule for function definitions, called the *rule of unfolding* (where $:=$ denotes substitution):

rule of unfolding: With f defined as above, we have for all values A and B:

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

□

examples: For every two functions f and g their *composition* can be defined in the language as a function h, as follows:

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

We can, however, also define a general-purpose composition function C with these functions for its parameters, such that $C \cdot f \cdot g$ is the composition of f and g, for all functions f and g:

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

□

0.3 function composition

The composition of functions f and g is written as $f \circ g$, that is, we have:

definition:

$$(f \circ g) \cdot x \;=\; f \cdot (g \cdot x) \quad , \text{ for all } f, g, x \quad .$$

□

The following little calculation demonstrates the *associativity* of function composition; each step in such a calculation is justified by a *hint*, written between braces $\{ \cdots \}$:

$$\begin{array}{ll} & ((f \circ g) \circ h) \cdot x \\ = & \quad \{ \text{ definition of } \circ \ \} \\ & (f \circ g) \cdot (h \cdot x) \\ = & \quad \{ \text{ definition of } \circ \ \} \\ & f \cdot (g \cdot (h \cdot x)) \\ = & \quad \{ \text{ definition of } \circ \ \} \\ & f \cdot ((g \circ h) \cdot x) \\ = & \quad \{ \text{ definition of } \circ \ \} \\ & (f \circ (g \circ h)) \cdot x \quad , \end{array}$$

from which it follows that $((f \circ g) \circ h) \cdot x$ equals $(f \circ (g \circ h)) \cdot x$, for all x; hence, we conclude:

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

Associativity of an operator is useful because it provides an opportunity for formula manipulation: for any associative operator $\oplus$ we may rewrite (sub)expressions of the shape $(x \oplus y) \oplus z$ as $x \oplus (y \oplus z)$ and vice versa.

0.4 extensionality

In the conclusion that $\circ$ is associative we have tacitly used that two functions are equal if all their applications are equal; this is called *extensionality* and is a direct consequence of the fact that all we can do with a function is apply it: functions all whose applications are equal cannot possibly be distinguished, so such functions are, to all intents and purposes, equal.

0.5 operator sections

For any binary operator $\oplus$, we use $(x \oplus)$, $(\oplus y)$, and $(\oplus)$ as abbreviations for the functions $\lambda y . x \oplus y$, $\lambda x . x \oplus y$, and $\lambda x \lambda y . x \oplus y$ respectively.

example: Having an explicit operator for function application, we can study $(f \cdot)$, $(\cdot x)$, and $(\cdot)$. We have $(f \cdot) = f$, and $(\cdot x)$ ("apply to x") is the function that applies its (function) argument to x. Moreover, $(\cdot)$ ("apply") is the (higher-order) function representing function application. If we apply Leibniz's rule to function $(\cdot x)$ we obtain as useful variation to this rule:

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

□

0.6 tuples

A *pair* of values x, y can also be considered as a single *composite* value; we denote this value as $\langle x, y \rangle$. More generally can we form arbitrary *tuples*: we also have, for instance, the unique empty tuple $\langle\rangle$, the singleton tuples $\langle x \rangle$ with elements x, triples $\langle x, y, z \rangle$, and so on.

0.7 lists

For any type B we use $\mathcal{L}_*(B)$ for the type of finite lists with elements in B. We use $[]$ ("empty") for the empty list and both $\triangleright$ ("cons") and $\triangleleft$ ("snoc") as list constructors: for element b and list x both $b \triangleright x$ and $x \triangleleft b$ are (usually different) lists. (For example: $[b,c,d] = b \triangleright [c,d]$ and $[b,c,d] = [b,c] \triangleleft d$.) The reason to use both $\triangleright$ and $\triangleleft$ is mainly aesthetical; for example, in an equation like $F \cdot x \cdot (\langle b \rangle \triangleright ss) = F \cdot (x \triangleleft b) \cdot ss$ the variables can now occur in the same order in both sides of the equation, and this enhances the readability of such an equation.

The symbol $*$ ("map") denotes the (binary) map-operator: for f of type $B \rightarrow C$ the function $(f*)$ has type $\mathcal{L}_*(B) \rightarrow \mathcal{L}_*(C)$, with:

$$
\begin{array}{lcl}
f*[] & = & [] \\
f*(b \triangleright x) & = & f\cdot b \triangleright f*x \\
f*(x \triangleleft b) & = & f*x \triangleleft f\cdot b
\end{array}
$$

0.8 an example

As a simple example of calculational reasoning, we show that $\sqrt{p}$ is irrational for every prime number p. For this purpose we use a function f, which we require to satisfy, for a fixed prime p and for all positive naturals x and y:

$$
\begin{array}{lcl}
f\cdot p & = & 1 \\
f\cdot(x*y) & = & f\cdot x + f\cdot y
\end{array}
$$

Informally, $f\cdot x$ is the number of prime factors p occurring in x. Of course, f is not completely defined by these two properties, but these are all we need in the proof. Now we derive, where dummies x, y range over the positive naturals:

$$
\begin{array}{ll}
 & \text{“}\sqrt{p}\text{ is rational”} \\
\equiv & \quad \{ \text{ definition of rationality } \} \\
 & (\exists x, y :: \sqrt{p} = x/y) \\
\equiv & \quad \{ \text{ algebra } \} \\
 & (\exists x, y :: p*y^2 = x^2) \\
\Rightarrow & \quad \{ \text{ Leibniz's rule, to introduce } f \} \\
 & (\exists x, y :: f\cdot(p*y^2) = f\cdot(x^2)) \\
\equiv & \quad \{ \text{ properties of } f \} \\
 & (\exists x, y :: 1+2*f\cdot y = 2*f\cdot x) \\
\equiv & \quad \{ \text{ all odd numbers differ from all even numbers } \} \\
 & \mathsf{false} \quad ,
\end{array}
$$

from which we conclude that $\sqrt{p}$ is irrational. Notice that we did not require x and y to be relatively prime, and that we have not used explicit mathematical induction: the latter is “hidden” in the postulated properties of f, and establishing these properties is a separate concern.

1 Implicit specifications

Sometimes a function specification is an *explicit* definition of that function, and sometimes such a definition already is an executable functional program. Actually, functional programming is often advocated as a proper tool for the construction of (so-called) executable specifications. Sometimes, however, functions are most conveniently specified *implicitly*, that is, by means of equations whose solutions are the functions thus specified. Here I wish to show, first, that (the most obvious) specifications of functional programs need not always be executable, and, second, that a calculational style of reasoning can be applied succesfully to solve such equations.

1.0 number conversion

In this example $\mathcal{L}_2$ and $\mathcal{L}_3$ denote the sets of finite lists with elements in $\{0,1\}$ and $\{0,1,2\}$, respectively. Furthermore, functions $v2$, of type $\mathcal{L}_2 \rightarrow \mathsf{Nat}$, and $v3$, of type $\mathcal{L}_3 \rightarrow \mathsf{Nat}$, are given, with the following definitions:

$$
\begin{array}{lcl@{\qquad}lcl}
v2\cdot[] & = & 0 & v3\cdot[] & = & 0 \\
v2\cdot(a \triangleright s) & = & a + 2*v2\cdot s & v3\cdot(b \triangleright t) & = & b + 3*v3\cdot t
\end{array}
$$

The problem now is to derive a definition for a function $c23$, of type $\mathcal{L}_2 \rightarrow \mathcal{L}_3$, and with specification:

$$
(\forall s :: v3\cdot(c23\cdot s) = v2\cdot s) \quad ,
$$

under the additional restriction that all integer (sub)expressions in the definition have *bounded* values. Informally, this specification states that $c23$ maps binary representations of natural numbers to *equivalent* ternary representations.

This specification can also be formulated more concisely at the function level as follows, but this is not always helpful:

$$
v3 \circ c23 = v2 \quad .
$$

We could, of course, also specify $c23$ *explicitly* in terms of the inverse of $v3$, as follows:

$$
c23 = v3^{-1} \circ v2 \quad ,
$$

but neither would this make the specification executable nor would it make the problem any easier.

Specifications of this shape are typical for (so-called) *representation changers*. The class of representation changers is quite large and comprises not only conversion routines but also parsers, compilers, and sorting algorithms.

1.1 solution

The general strategy is that an equation –with x as the unknown– of the shape $x : f\cdot x = E$ can be solved by rewriting the right-hand side E into an equivalent expression of the form $f\cdot F$; the equation thus becomes $x : f\cdot x = f\cdot F$, from which we conclude that $x = F$ solves the equation. For our example problem we must, to obtain a definition for $c23\cdot s$, solve the equation $x : v3\cdot x = v2\cdot s$, so we must rewrite $v2\cdot s$ into an expression of the shape $v3\cdot(\cdots)$. Because of the case analysis in $v2$'s definition, which must be used anyhow, we perform such rewriting for two cases, with induction on (the length of) s:

$$
\begin{array}{ll}
 & v2\cdot[] \\
= & \quad \{ \ v2 \ \} \\
 & 0 \\
= & \quad \{ \ v3 \ \} \\
 & v3\cdot[] \quad ,
\end{array}
$$

from which we conclude that $c23\cdot[] = []$ is a solution, and:

$$\begin{array}{ll} & v2\cdot(a \triangleright s) \\ = & \quad \{\ v2\ \} \\ & a + 2*v2\cdot s \\ = & \quad \{\ \text{Induction Hypothesis, to introduce } v3 \text{ and to eliminate } v2\ \} \\ & a + 2*v3\cdot(c23\cdot s) \\ = & \quad \{\ \text{introduction of function } f \text{ (see below)}\ \} \\ & v3\cdot(f\cdot a\cdot(c23\cdot s)) \quad , \end{array}$$

from which we conclude that $c23\cdot(a \triangleright s) = f\cdot a\cdot(c23\cdot s)$ is acceptable.

The decision to introduce a new function, f, is not at all far-fetched: the expression $a + 2*v3\cdot(c23\cdot s)$ neither matches the right-hand-side of $c23$'s specification, nor is it a simple generalization thereof; hence, the only thing we can do is introduce a new function to bridge the gap between this expression and our target, namely an expression of the shape $v3\cdot(\cdots)$.

For the above to be correct, function f, of type $\{0,1\} \rightarrow \mathcal{L}_3 \rightarrow \mathcal{L}_3$, must satisfy:

$$(\forall a, t :: v3\cdot(f\cdot a\cdot t) = a + 2*v3\cdot t) \quad .$$

As a result of the above derivations we now obtain as definition for $c23$, in terms of this f:

$$\begin{array}{lcl} c23\cdot[] & = & [] \\ c23\cdot(a \triangleright s) & = & f\cdot a\cdot(c23\cdot s) \end{array}$$

To obtain a definition for f we derive, following the same strategy and with induction on t:

$$\begin{array}{ll} & a + 2*v3\cdot[] \\ = & \quad \{\ v3\text{, algebra}\ \} \\ & a \\ = & \quad \{\ \text{algebra, } v3\ \} \\ & a + 3*v3\cdot[] \\ = & \quad \{\ v3\ \} \\ & v3\cdot[a] \quad , \end{array}$$

and:

$$\begin{array}{ll} & a + 2*v3\cdot(b \triangleright t) \\ = & \quad \{\ v3\ \} \\ & a + 2*(b + 3*v3\cdot t) \\ = & \quad \{\ \text{algebra}\ \} \\ & a + 2*b + 2*3*v3\cdot t \\ = & \quad \{\ c = c\,\mathsf{mod}\,3 + 3*(c\,\mathsf{div}\,3)\text{, with } c = a + 2*b\ \} \\ & c\,\mathsf{mod}\,3 + 3*(c\,\mathsf{div}\,3) + 2*3*v3\cdot t \\ = & \quad \{\ \text{algebra}\ \} \\ & c\,\mathsf{mod}\,3 + 3*(c\,\mathsf{div}\,3 + 2*v3\cdot t) \\ = & \quad \{\ \text{Induction Hypothesis } (0 \le c < 6\text{, hence } 0 \le c\,\mathsf{div}\,3 < 2)\ \} \\ & c\,\mathsf{mod}\,3 + 3*(v3\cdot(f\cdot(c\,\mathsf{div}\,3)\cdot t)) \\ = & \quad \{\ v3\ \} \\ & v3\cdot(c\,\mathsf{mod}\,3 \triangleright f\cdot(c\,\mathsf{div}\,3)\cdot t) \quad . \end{array}$$

Thus we obtain as a definition for f:

$$\begin{array}{lcl} f\cdot a\cdot[] & = & [a] \\ f\cdot a\cdot(b \triangleright t) & = & c\,\mathsf{mod}\,3 \triangleright f\cdot(c\,\mathsf{div}\,3)\cdot t \\ & & \mathsf{where}\ \ c = a + 2*b\ \ \mathsf{end} \end{array}$$

(If so desired, nonsignificant leading zeroes can be avoided by refining the first case of this definition into $f\cdot 0\cdot[] = []$ and $f\cdot 1\cdot[] = [1]$.)

2 Generalization by abstraction

An important problem solving technique is *generalization*. Most problems admit very many generalizations, so it is important to have a way to identify the useful ones. 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 more general expression may be the generalization we are looking for. Put differently, generalizations can be found more easily by abstraction from the differences between two (or more) similar expressions rather than by obtaining inspiriation from just a single expression.

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 especially when the 1 arose 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.

□

2.0 folded operators

To illustrate generalization by abstraction we reinvent the concept of *folded operators*, which are discussed more extensively in R.S. Bird and Ph. Wadler's textbook [0]. We use this to obtain a theorem for the transformation of *linear recursion* into *tail recursion*. Moreover, we shall apply this theorem in Section 3.1.

We consider a binary operator $\oplus$ of type $B \times U \rightarrow U$, for some types B and U. In this section dummies b and c have type B whereas u has type U. With $\oplus$ we can form expressions of type U, like:

$$u \quad , \quad b \oplus u \quad , \quad b \oplus (c \oplus u) \quad \text{, and so on.}$$

These expressions have in common that they depend on *one* value of type U 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 U and a single $\mathcal{L}_*(B)$, by introducing a binary operator $\otimes$, of type $\mathcal{L}_*(B) \times U \rightarrow U$, thus:

$$\begin{array}{rcl} u & = & [] \otimes u \\ b \oplus u & = & [b] \otimes u \\ b \oplus (c \oplus u) & = & [b,c] \otimes u \end{array}$$

Moreover, because $b \oplus (c \oplus u) = (b \oplus u)(u := c \oplus u)$, our new operator must, for the sake of consistency, also satisfy:

$$[b,c] \otimes u \;=\; [b] \otimes (c \oplus u) \quad .$$

By a simple generalization –replace the singleton list $[b]$ by any list x– we obtain the following recursive definition for $\otimes$:

$$\begin{array}{rcl} [] \otimes u & = & u \\ (x \lhd c) \otimes u & = & x \otimes (c \oplus u) \end{array}$$

Operator $\otimes$ has other interesting properties, but we shall not need these here; $\otimes$ is well-known: it is called *foldr* $(\oplus)$ in Bird and Wadler's text, but in calculations I prefer a (short) name like $\otimes$ over a (long) expression like *foldr* $(\oplus)$. The above shows that such *folded operators* can be invented without operational interpretations, 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 $B := (U \rightarrow U)$ function application indeed has type $B \times U \rightarrow U$. So, we now introduce $\odot$ for folded $\cdot$, and we obtain as definition:

$$\begin{array}{rcl} [] \odot u & = & u \\ (x \lhd g) \odot u & = & x \odot (g \cdot u) \end{array} \quad ,$$

with properties like:

$$[f,g] \odot u \;=\; f \cdot (g \cdot u) \quad .$$

Hence, a list x of type $\mathcal{L}_*(U \rightarrow U)$ now represents the continued composition of its (function) elements and $\odot$ amounts to continued function application: $x \odot u$ is the application to u of the functions in x. (Moreover, this example shows that it really helps to have an explicit symbol for function application.)

□

2.1 from linear to tail recursion

We consider the following recursive definition of a function F, in terms of given functions b, f, g and h:

$$\begin{array}{rcl} F \cdot x & = & \textsf{if} \;\; \neg b \cdot x \rightarrow f \cdot x \\ & & [] \;\; b \cdot x \rightarrow h \cdot x \cdot (F \cdot (g \cdot x)) \\ & & \textsf{fi} \end{array}$$

Function F has type $X \rightarrow U$, for some types X and U, provided b, f, g, and h have the following types:

$$\begin{array}{l} b \;:\; X \rightarrow \textsf{Bool} \\ f \;:\; X \rightarrow U \\ g \;:\; X \rightarrow X \\ h \;:\; X \rightarrow U \rightarrow U \end{array}$$

Function h has two arguments, so we can also write it as a binary operator $\oplus$, of type $X \times U \rightarrow U$:

$$x \oplus u \;=\; h \cdot x \cdot u \quad \text{, for all } x, u \quad ,$$

and we can adapt the definition of F accordingly:

$$\begin{array}{rcl} F \cdot x & = & \textsf{if} \;\; \neg b \cdot x \rightarrow f \cdot x \\ & & [] \;\; b \cdot x \rightarrow x \oplus F \cdot (g \cdot x) \\ & & \textsf{fi} \end{array}$$

With $\otimes$ for the folded version of $\oplus$ 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:

$$\begin{array}{rcl} & & s \otimes F \cdot x \\ & = & \quad \{ \; F, \text{ case } \neg b \cdot x \; \} \\ & & s \otimes f \cdot x \end{array} \quad ,$$

and:

$$\begin{array}{rcl} & & s \otimes F \cdot x \\ & = & \quad \{ \; F, \text{ case } b \cdot x \; \} \\ & & s \otimes (x \oplus F \cdot (g \cdot x)) \\ & = & \quad \{ \otimes \} \\ & & (s \lhd x) \otimes F \cdot (g \cdot x) \end{array} \quad .$$

We now introduce a function G with the following specification:

$$G{\cdot}s{\cdot}x \;=\; s \otimes F{\cdot}x \quad \text{, for all } s, x \quad .$$

This is a generalization of F and, as a result of the above derivation, we obtain the following theorem.

tail-recursion theorem:

if:
$$F{\cdot}x \;=\; \textsf{if}\ \neg b{\cdot}x \rightarrow\ f{\cdot}x \ \ [\!]\ \ b{\cdot}x \rightarrow\ x \oplus F{\cdot}(g{\cdot}x)\ \ \textsf{fi}$$

then:
$$F{\cdot}x \;=\; G{\cdot}[\,]{\cdot}x$$

where:
$$G{\cdot}s{\cdot}x \;=\; \textsf{if}\ \neg b{\cdot}x \rightarrow\ s \otimes f{\cdot}x \ \ [\!]\ \ b{\cdot}x \rightarrow\ G{\cdot}(s \lhd x){\cdot}(g{\cdot}x)\ \ \textsf{fi}$$

and:
$$[\,] \otimes u \;=\; u$$
$$(s \lhd x) \otimes u \;=\; s \otimes (x \oplus u)$$

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 viewed as the "stack" that is traditionally used to implement recursion. Because tail-recursion can be implemented by simple iteration, 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, the list parameter s is represented by an array variable c and an integer p, according to the representation invariant:

$$s \;=\; c(i : 0 \leq i < p) \;\wedge\; 0 \leq p \quad .$$

program:

$$\begin{array}{l} \quad p, x := 0, x0 \\ ;\ \{\,\textsf{invariant:}\ \ F{\cdot}x0 = G{\cdot}s{\cdot}x\,\} \\ \quad \textsf{do}\ \ b{\cdot}x \ \rightarrow\ \ c[p] := x\ ;\ p := p{+}1\ ;\ x := g{\cdot}x\ \ \textsf{od} \\ ;\ \{\,F{\cdot}x0 = s \otimes f{\cdot}x\,\} \\ \quad u := f{\cdot}x \\ ;\ \{\,\textsf{invariant:}\ \ F{\cdot}x0 = s \otimes u\,\} \\ \quad \textsf{do}\ \ p{\neq}0 \ \rightarrow\ \ p := p{-}1\ ;\ u := c[p] \oplus u\ \ \textsf{od} \\ \quad \{\,F{\cdot}x0 = u\,\} \end{array}$$

□

2.2 tail 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.

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.)

□

Fusion is particularly simple if the definition of F is tail-recursive. This is the prototype of a tail-recursive definition:

$$F{\cdot}x \;=\; \textsf{if}\ \neg b{\cdot}x \rightarrow\ f{\cdot}x \ \ [\!]\ \ b{\cdot}x \rightarrow\ F{\cdot}(g{\cdot}x)\ \ \textsf{fi}$$

¿From this definition we obtain an equally tail-recursive definition for $G \circ F$, simply by replacing the base case, $f{\cdot}x$, by $G{\cdot}(f{\cdot}x)$:

$$H \;=\; G \circ F$$

$$H{\cdot}x \;=\; \textsf{if}\ \neg b{\cdot}x \rightarrow\ G{\cdot}(f{\cdot}x) \ \ [\!]\ \ b{\cdot}x \rightarrow\ H{\cdot}(g{\cdot}x)\ \ \textsf{fi}$$

Similarly, the above definition of F can itself be considered as the result of fusing f with the (slightly) simpler function F' defined by:

$$F'{\cdot}x \;=\; \textsf{if}\ \neg b{\cdot}x \rightarrow\ x \ \ [\!]\ \ b{\cdot}x \rightarrow\ F'{\cdot}(g{\cdot}x)\ \ \textsf{fi}$$

Tail-fusion does not always yield more efficient programs, but recognizing that a definition is a fusion result enhances clarity, as this makes explicit that the function is

the composition of other functions, for example, as in $F = f \circ F'$. Thus, "unfusion" contributes to our insight in the function's structure.

Finally, because function composition is associative, tail-fusion can be repeated as often as desired: the base case $G \cdot (f \cdot x)$ equals $(G \circ f) \cdot x$, so it matches the pattern of the prototype again.

3 Case study: expression evaluation

Expressions are linear strings of symbols representing tree structures. Evaluators, which are mechanisms for computing the "values" of such trees, 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 than the tree it represents.

¿From a definition of trees as a starting point, an evaluator can be designed by first *choosing* a linear representation and next 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. This example also suggests that expression parsing could well do without the grammars-and-languages paradigm. The example is about simple expressions, which either are *primitive* terms –like constants, variables, or function applications–, or expressions *composed* from other ones by means of binary operators.

3.0 trees and their values

In what follows, dummy b has type B and dummy c has type C, for some *disjoint* types B and C, so we have $(\forall b, c :: b \neq c)$. The datatype T of trees is defined (recursively) as follows:

$$\begin{array}{lll} \langle b \rangle & \in T & \text{, for all } b \\ \langle s, c, t \rangle & \in T & \text{, for all } c \text{ and } s, t \in T \ . \end{array}$$

Next, we assume that, for some fixed type M, functions f and g have been given, with the following types:

$$\begin{array}{l} f \in B \rightarrow M \\ g \in C \rightarrow M \rightarrow M \rightarrow M \end{array}$$

Type M is the type of the values of trees, as is reflected by the following definition of function $V \in T \rightarrow M$, with the intention that $V \cdot s$ be the value of tree s:

$$\begin{array}{lcl} V \cdot \langle b \rangle & = & f \cdot b \\ V \cdot \langle s, c, t \rangle & = & g \cdot c \cdot (V \cdot s) \cdot (V \cdot t) \end{array}$$

3.1 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. The main design decision now is that we study $V* ss$, for ss of type $\mathcal{L}_*(T)$, for two reasons. First, it is a generalization, because $[V \cdot s] = V*[s]$, and, second, $V \cdot s$ and $V \cdot t$ are the elements of the list $V*[s, t]$; thus, we try to combine the two recursive applications of V into a single one.

As always we have $V*[\,] = [\,]$; furthermore, in compliance with the case analysis in V's definition, we derive:

$$\begin{array}{ll} & V*(\langle b \rangle \triangleright ss) \\ = & \quad \{ \ * \ \} \\ & V \cdot \langle b \rangle \triangleright V* ss \\ = & \quad \{ \ V \ \} \\ & f \cdot b \triangleright V* ss \\ = & \quad \{ \text{ introduction of } \ominus \text{ (motivation follows, see below) } \} \\ & b \ominus V* ss \quad , \end{array}$$

and:

$$\begin{array}{ll} & V*(\langle s, c, t \rangle \triangleright ss) \\ = & \quad \{ \ * \ \} \\ & V \cdot \langle s, c, t \rangle \triangleright V* ss \\ = & \quad \{ \ V \ \} \\ & g \cdot c \cdot (V \cdot s) \cdot (V \cdot t) \triangleright V* ss \\ = & \quad \{ \text{ eureka! introduction of } \oplus \text{ (see below) } \} \\ & c \oplus (V \cdot s \triangleright V \cdot t \triangleright V* ss) \\ = & \quad \{ \ * \text{ (twice) } \} \\ & c \oplus (V*(s \triangleright t \triangleright ss)) \quad . \end{array}$$

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 all far-fetched: the only way to stay within the pattern –expressions of the shape $V*(\cdots)$– is to collect $V \cdot s$, $V \cdot t$, and $V* ss$ into $V*(s \triangleright t \triangleright ss)$. Notice

that the double use of $\oplus$ is harmless: 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 $(V*)$:

$$\begin{array}{lcl} V*\,[\,] & = & [\,] \\ V*\,(\langle b\rangle \triangleright ss) & = & b \oplus (V*\,ss) \\ V*\,(\langle s,c,t\rangle \triangleright ss) & = & c \oplus (V*\,(s \triangleright t \triangleright ss)) \end{array}$$

Now we apply the tail-recursion theorem from Section 2.1 to transform this linear recursion into tail recursion. With $\otimes$ for 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 $(V*)$ can be defined in terms of a function G as follows:

$$V*\,ss \;=\; G{\cdot}[\,]{\cdot}ss \quad .$$

where function G, of type $\mathcal{L}_*(B \cup C) \to \mathcal{L}_*(T) \to \mathcal{L}_*(M)$, has as specification:

$$G{\cdot}x{\cdot}ss \;=\; x \otimes V*\,ss \quad .$$

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:

$$[V{\cdot}s] \;=\; G{\cdot}[\,]{\cdot}[s] \quad .$$

* * *

According to the rule for tail-fusion –cf. Section 2.2– we can view G as the composition of $(\otimes[\,])$ and a function F:

$$G{\cdot}x{\cdot}ss \;=\; F{\cdot}x{\cdot}ss \otimes [\,] \quad .$$

where a definition for F is obtained from G's definition by simply omitting $\otimes\,[\,]$. Summarizing, we obtain the following set of definitions:

$$\begin{array}{lcl} [V{\cdot}s] & = & F{\cdot}[\,]{\cdot}[s] \otimes [\,] \\ \\ F{\cdot}x{\cdot}[\,] & = & x \\ F{\cdot}x{\cdot}(\langle b\rangle \triangleright ss) & = & F{\cdot}(x \triangleleft b){\cdot}ss \\ F{\cdot}x{\cdot}(\langle s,c,t\rangle \triangleright ss) & = & F{\cdot}(x \triangleleft c){\cdot}(s \triangleright t \triangleright ss) \\ \\ [\,] \otimes ms & = & ms \\ (x \triangleleft b) \otimes ms & = & x \otimes (f{\cdot}b \triangleright ms) \\ (x \triangleleft c) \otimes (m \triangleright n \triangleright ms) & = & x \otimes (g{\cdot}c{\cdot}m{\cdot}n \triangleright ms) \end{array}$$

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:

$$\begin{array}{lcl} [\,] \otimes ss & = & ss \\ (x \triangleleft b) \otimes ss & = & x \otimes (\langle b\rangle \triangleright ss) \\ (x \triangleleft c) \otimes (s \triangleright t \triangleright ss) & = & x \otimes (\langle s,c,t\rangle \triangleright ss) \end{array}$$

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.

References

[0] R.S. Bird, Ph. Wadler, *Introduction to functional programming.*
Prentice Hall International, Hemel Hempstead, 1988.

[1] E.W. Dijkstra, C.S. Scholten, *Predicate calculus and program semantics.*
Springer-Verlag, New York, 1990.

[2] R.R. Hoogerwoord, *On the foundations of functional programming: a programmer's point of view.*
Computing Science Note 94/26, Eindhoven University of Technology, 1994.