29 Years of (Functional) Programming: what have we learned?

If it is clumsy, it is not mathematics.
Edsger W. Dijkstra

0 A little bit of history

On 1 September 1984, I began to work in the Department of Mathematics and Computing Science as an Universitair Docent, under the supervision of full professor Martin Rem. Although my appointment included tenure, Martin Rem insisted, from the very beginning, that it would be in my own interest that I should obtain a doctoral degree, so I should write a PhD-thesis.

After over half a year of initial deliberation I decided that Functional Programming would be the subject of my PhD-thesis, for the following reasons:

- The subject matched well with my interest in a more mathematical approach to programming in general, and ...
- More specifically, Functional Programming deals with (functional) relations between values to be computed, thus abstracting from the execution order prevailing in imperative programming.
- The, somewhat opportunistic, consideration that it would be a field from which results could be reaped fairly easily, that is, within a reasonable amount of time.
- The lack of competition, at least within our University, because at that time nobody else seemed to study this subject.

In addition, the Study Guide for our department announced a course on Functional Programming, but until that moment this course had no teacher. So, I decided that I would set up such a course and teach it, being fully aware that the best way to master a subject is to teach about it.

* * *

In imperative programming we had the standard technique of *replacing a constant by a variable*, to obtain a candidate for a repetition invariant For about 5 years we already knew that this was not effective for all problems. For some problems the use of a (so-called) *tail invariant* was more appropriate, but it was difficult to decide, in an early stage of the design, which of the two techniques should be used. Replacing a constant by variable is simpler, whereas tail invariants seemed to be more general.

Today we know why: the decision to replace a constant by a variable actually is *premature*. Before deciding on whatever shape the invariant should have, a programmer should first investigate what are the relations between the values to be computed. Here functional programming enters the picture, as (functional) relations between values is its subject.

Thus, functional programming can be used in two ways: either as an activity in its own right, where the programs will executed directly by means of a compiler for the functional-programming language, or as a stepping stone in the process of constructing

imperative programs. In the latter case, functional programming contributes effectively to an important *Separation of Concerns*: on the one hand, one has to decide what the relations are between the values be computed; on the other hand, one has to decide in what order these values will be computed. The important observation is here that these two concerns can be, and therefore should be, separated.

In this lecture I will illustrate a few of the more commonly used programming techniques. I will do so by means of extremely simple examples, almost trivial ones; please, keep in mind, however, that my subject today is the techniques, not the examples themselves.

1 The Importance of Naming

Experience has shown that it really helps to give a name to relatively simple reasoning principles or designing principles, so simple actually, that at first sight they do not seem to deserve to be named at all!

The point is, however, that by giving a name to such a principle anyhow, the principle becomes easier to remember and easier to apply explicitly; that enhances such a principle's effectiveness.

In the following subsections we illustrate this with a few examples.

1.0 Leibniz's Principle of Equals for Equals

Everybody who has acquired the rules of arithmetic in primary school will accept (the correctness of) the following little calculation:

$$\begin{array}{ll} & (2+3)*4 \\ = & \quad \{\ 2+3=5\ \} \\ & 5*4\ . \end{array}$$

What is really going on here? We have a composite expression with a subexpression "$(2+3)$". The value of this subexpression equals another, simpler, subexpression, namely "5", and in the composite expression we have *replaced* one subexpression by another subexpression having the same value. And this replacement, so we conclude, does not change the value of the expression as a whole.

The great mathematician Gottfried Wilhelm Leibniz was the first to recognize the importance of this principle and to formulate it explicitly: replacing, in a composite expression, (an occurrence of) a subexpression by an equivalent subexpression does not affect the value of the composite expression. This is a way to formulate that the value of a composite expression depends on the values of its subexpressions *only*, and not on other aspects of these subexpressions (like, for example, the number of symbols in them).

Every composite expression is a function of its subexpressions. If we now abstract from the more syntactical aspects of this, we find that Leibniz's principle boils down to the property that function application *preserves equality*. As a matter of fact, this is the *only* general property of function application. So, more abstractly, Leibniz's principle can be formulated as follows, where f denotes any function and for all x and y in that function's domain:

$$(0) \qquad x=y \ \Rightarrow\ (\forall f ::\ f\cdot x = f\cdot y)\ .$$

There is more to it, though. The operation of applying any function to a fixed argument itself is a function application in which that "any function" now is the argument. Put differently, function application is a binary operator, that is, a two-argument function, in which the function argument can be treated as any other argument. Thus, we obtain a useful variant of Leibniz's principle, for functions f and g with the same domain:

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

By straightforward logical contraposition, and by means of the rules of De Morgan, rules (0) and (1) can be rewritten into the following, equivalent forms:

(2) $\quad x \neq y \;\Leftarrow\; (\exists f :: f \cdot x \neq f \cdot y)$.

(3) $\quad f \neq g \;\Leftarrow\; (\exists x :: f \cdot x \neq g \cdot x)$.

In words, rule (2) states that a sufficient condition to prove that two values are different is to show the existence of a function that has different values for the two given values. This is useful if the range of the function offers better possibilities to prove differences than the function's domain. Similarly, rule (3) states that a sufficient condition to prove that two functions are different is to show the existence a value in the common domain of the functions where the two given functions have a different value.

* * *

One may well wonder why so much attention is paid to so simple a matter. Experience shows that the explicit formulations, in the form of (0) through (3), of Leibniz's principle certainly helps: by being aware of these formulations opportunities for their application are identified more easily. In Section 2 we will encounter several examples of this.

As far as I know, it has been Edsger W. Dijkstra who has begun to use the name "Leibniz" to refer explicitly to, particularly, rule (0). This was in the early 1980s. Wim Feijen has suggested to refer to variant (2) by "Zinbiel". I myself use "Leibniz" to refer to any the four variants. To what extent Dijkstra was aware of variant (1) I do not know.

Remark: Depending on one's field of interest, Leibniz's principle lives under several different guises, all of which amount to the very same principle:

Leibniz's rule of equals for equals
substitutivity
congruence relations
homomorphisms
compositionality
referential transparency

□

1.1 Cantor's Diagonalization Principle

In the predicate calculus we have the following two rules of *instantiation*. For a given set V and for a given predicate P on V we have, for all $y \in V$ –the condition $y \in V$ is essential here–:

(4) $\quad (\forall x : x \in V : P(x)) \;\Rightarrow\; P(y)$.

(5) $\quad (\exists x : x \in V : P(x)) \Leftarrow P(y)$.

These rules are called "instantiation" because, when applied from right to left, a quantor is removed from the formula, instantiated, if you like. But these rules may also be applied from left to right, of course, in order to *introduce* a quantor.

In the Natural Deduction style of reasoning, rule (4) is usually applied from right to left only, and then is called $\forall$-*elimination*. Also, rule (5) is usually applied from left to right only, and then is called $\exists$-*introduction*. Both rules are faces of the same coin, however, because they are each other's duals, on account of De Morgan's rules.

* * *

Now we consider the following pattern of reasoning, for some given set V and for some given two-place predicate Q on $V \times V$:

$$
\begin{array}{ll}
 & (\forall x : x \in V : (\forall y : y \in V : Q(x,y)\,)\,) \\
\Rightarrow & \quad \{ \text{ instantiation } y := x, \text{ using that } x \in V \ \} \\
 & (\forall x : x \in V : Q(x,x)\,) \quad .
\end{array}
$$

If simplification is our goal, and if we have no further knowledge about set V –or if we do not wish to use such knowledge at this stage–, then the step "instantiation $y := x$" is about the only thing we can do. After all, in the scope of the universal quantification "$\forall x$", the only value in set V we know its existence of is x, so the "instantiation $y := x$" is the only possibility here!

So, why give such a simple reasoning step a name? Yet, this is the essence of what is known as "Cantor's Diagonalization Principle", that is, what remains of it after we abstract from the specific details that usually come with it. As a reasoning technique, it has been attributed to the mathematician Georg Cantor, after whom the principle has been named. However trivial the principle may be from a logical point of view, it has proved to be very effective, and as such it certainly deserves its name. It plays a central role in several important theorems, like Cantor's theorem on the cardinality of power sets, Gödel's incompleteness result, and the solution to the Halting Problem.

1.2 Generalization by Abstraction

Generalization is to consider a given problem as a *specific instance* of a whole class of similar problems, by *deliberately ignoring* some detail(s) of the problem.

For example, ask an arbitrary mathematician how he would compute the 137-th prime number, and he will immediately start thinking about the problem how to compute the n-th prime number, for any natural number n. This is a generalization and it is meaningful, for two reasons.

Firstly, the number 137 in the original problem probably is not very relevant: computing the 137-th prime number may be expected to be neither easier nor more difficult than computing the 136-th or 138-th prime number, or any other prime number, for that matter. So, we had better ignore this particular number and generalize the problem, as we may expect that this will not make the problem more difficult.

Secondly, by replacing 137, which is a constant, by n, which is a variable, we open up the possibility to formulate useful relations between, say, the n-th prime number and the

prime numbers with lower indices. Thus, our manipulative freedom, our wriggling room, if you like, is enlarged, and this may even make the problem easier.

Not all forms of generalization are equally viable, though. No mathematician in his right mind, for example, will generalize the problem of the 137-th prime number by thinking about the 137-th element of any arbitrary infinite sequence of natural numbers: this is a generalization all-right, but the set of all infinite sequences of natural numbers is way to chaotic to be of any use.

* * *

What makes generalization difficult is that any non-trivial problem allows for many meaningful generalizations; so, we are faced with an *embarras du choix* here: how to choose the "right" one to start with? Well, it helps very much if we have *several* – more than one – different expressions that nevertheless have enough in common to ponder the question: what is the common pattern behind these expressions? That is, can we view these *different* expressions as special instances of a *single* more general expression?

This principle I have called "Generalization by Abstraction". The principle is that useful generalizations may be identified by abstracting from the differences between several, different but similar, expressions. Viewing these expressions as instances of a more abstract, more general expression often yields useful generalizations.

As in the previous examples, the idea is simple, hardly deserving to be given a name, but it has turned out to be remarkably effective. Because this principle has played a central role in my work on Functional Programming, I will devote a whole section to it, later in this lecture. Here I will only illustrate the principle by means of an extremely simple example.

Example: What do 0 and 1 have in common? That they are natural numbers! Of course, they also have in common that they are 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 set of 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 Calculemus!

An as yet insufficiently recognized but undeniable development is the emergence of a more calculational style of problem solving, applicable to the construction of both programs and mathematical proofs in general. In this style symbols and formulae play a much more important role than in traditional mathematics. The advantages of such a calculational style are:

- The shape of the formulae provides heuristic guidance.
- Assumptions and other properties used are made explicit,

- and so are design decisions.
- Thus, the mathematical structure of the design is clarified.

We illustrate this by means of a few relatively simple examples. Yet, these examples illustrate, not only the effectiveness of calculational reasoning, but also that calculational reasoning enhances the discovery of properties needed, and of modularization. Thus, fewer rabbits need to be pulled out of the magician's hat.

2.0 A trivial example

We consider a set Ω on which we have a binary infix operator "$\cdot$" ("dot"). So, this operator has type $\Omega \times \Omega \to \Omega$. To save parentheses we adopt the (syntactic) convention that this operator is *left-binding*; that is, $x \cdot y \cdot z$ must be parsed as $(x \cdot y) \cdot z$.

We now assume that Ω contains elements I and K – this implies, of course, that Ω is non-empty –, with the following properties, for all $x, y \in \Omega$:

(6) $\quad I \cdot x \;=\; x$,
(7) $\quad K \cdot x \cdot y \;=\; x$.

As an example of how we can derive properties of I and K in a calculational way, we prove here that I and K are different. Notice that this proof is completely of the type "there is only one thing you can do":

$I \neq K$
$\Leftarrow$ { Leibniz, to prepare for use of (6) (and (7)) }
$(\exists x :: I \cdot x \neq K \cdot x)$
$\Leftrightarrow$ { definition (6), of I }
$(\exists x :: x \neq K \cdot x)$
$\Leftarrow$ { Leibniz once more, to prepare for use of (7) }
$(\exists x :: (\exists y :: x \cdot y \neq K \cdot x \cdot y))$
$\Leftrightarrow$ { definition (7), of K }
$(\exists x :: (\exists y :: x \cdot y \neq x))$
$\Leftarrow$ { instantiation $x := I$, using $I \in \Omega$, (the simplest way) to get rid of $x \cdot y$ }
$(\exists y :: I \cdot y \neq I)$
$\Leftrightarrow$ { definition (6), of I }
$(\exists y :: y \neq I)$
$\Leftarrow$ { instantiation $x := I$ }
$(\forall x :: (\exists y :: y \neq x))$.

Until now, all that has been given about Ω are properties (6) and (7). Both of these express *equalities* and, therefore, without further knowledge it is impossible to prove a formula like $(\forall x :: (\exists y :: y \neq x))$, which is about *differences*. This formula expresses that our universe Ω is *not a singleton set*. As far as (6) and (7) are concerned Ω *might* be a singleton set, with everything being equal to everything. In such a model it would be impossible, of course, to prove $I \neq K$. So the above calculation shows that $I \neq K$ follows from the additional assumption:

(8) $(\forall x :: (\exists y :: y \neq x))$,

which formally expresses that Ω is *not a singleton set*. Notice that, in combination with the knowledge that Ω is non-empty, this is equivalent to the proposition that Ω has *at least 2 elements*.

In a very similar way it is also possible to prove:

(9) $(\forall x :: I \neq K \cdot x)$.

2.1 A not-so-trivial example

This is all very simple and straightforward, but now it is also possible to prove, from the above three assumptions alone, that our universe Ω is *infinite*, a fairly non-trivial result indeed.

What does it mean that a set is infinite? I only know of two different ways to define this:

- A set is infinite if every finite subset of it is not the whole set.
- A set is infinite if the set "contains" the natural numbers.

These two characterizations are not as different as they may seem: the first one is in terms of *finite* sets, and it is virtually impossible to discuss finite sets without introducing the natural numbers. Mind you, here we are computing engineers, not philosophers nor involved with the foundations of mathematics: we may safely take the natural numbers for granted.

The first characterization, for example, is used in proofs that the set of prime numbers is infinite. For the proof that our set Ω is infinite we will use the second one.

In order to do so we must be more precise about the term "contain". Our set Ω "contains" the natural numbers means that there is an *injective function* from the natural numbers into Ω. In plain English this means: Ω contains an *infinite sequence* all whose elements are *different*.

* * *

So, in Ω we must construct an infinite sequence s, which is just a function of type $\mathbb{N} \to \Omega$, which is injective:

(10) $(\forall i, j : 0 \leq i < j : s_i \neq s_j)$.

One way to define such a sequence is by means of recursion. That is, we choose values $X \in \Omega$ and $F \in \Omega$, and using these we define our sequence s as follows:

(11) $s_0 = X \ \wedge \ (\forall i : 0 \leq i : s_{i+1} = F \cdot s_i)$.

Now, by means of straightforward Mathematical Induction requirement (10) can be proved, under the additional assumption that X and F have the following two properties; note that these properties do not have to be pulled out of the magician's hat, they just *emerge* from the attempt to prove (10):

(12) $(\forall x : x \in \Omega : X \neq F \cdot x)$, and:

(13) $(\forall x, y : x, y \in \Omega : x \neq y \Rightarrow F \cdot x \neq F \cdot y)$.

These two properties constitute the *specifications* of X and F: if these are satisfied then sequence s, as defined by (11), satisfies requirement (10). This embodies a nice Separation of Concerns: (12) and (13) are requirements on X and F, which are values in Ω, in which the natural numbers do not occur anymore.

All we have to do now is define X and F in such a way that we can prove (12) and (13). We propose – see the Remarks, below –:

(14) $X = I$, and:

(15) $F = K$.

We prove (12) as follows, for all $x \in \Omega$:

$$
\begin{array}{ll}
 & X \neq F \cdot x \\
\Leftrightarrow & \quad \{ \text{ definitions (14), of } X, \text{ and (15), of } F \} \\
 & I \neq K \cdot x \\
\Leftrightarrow & \quad \{ \text{ property (9) } \} \\
 & \mathsf{true} \ ,
\end{array}
$$

and we prove (13), for all $x, y \in \Omega$:

$$
\begin{array}{ll}
 & F \cdot x \neq F \cdot y \\
\Leftrightarrow & \quad \{ \text{ definition (15), of } F \} \\
 & K \cdot x \neq K \cdot y \\
\Leftarrow & \quad \{ \text{ Leibniz, to prepare for use of (7) } \} \\
 & (\exists z :: K \cdot x \cdot z \neq K \cdot y \cdot z \,) \\
\Leftrightarrow & \quad \{ \text{ definition (7), of } K \} \\
 & (\exists z :: x \neq y \,) \\
\Leftrightarrow & \quad \{ \, \Omega \text{ is non-empty } \} \\
 & x \neq y \ .
\end{array}
$$

Remarks: A more traditional mathematical proof of (10) would be presented in a more *bottom-up* fashion. That is, after having established the need to prove (10), definitions (11), (14), and (15) would be pulled out of the hat right-away, and then a proof of (10) would be given.

The above proof is more *top-down*, more *demand driven*, if you like. Definition (11) is sweetly reasonable, yet it represents a design decision. By *not* rushing into choosing definitions for X amd F, and by first trying to prove (11) instead, we obtain information on what is needed. The resulting specifications (12) and (13) provide a useful Separation of Concerns: the unavoidable Mathematical Induction now has been dealt with, and the natural numbers, having played their role, can leave the stage again.

Regarding the choice of X and F, in the form of definitions (14) and (15): our knowledge of Ω is rather limited: we do know that $I \in \Omega$ and $K \in \Omega$, and that we can form more complicated values, like $I \cdot I$, $I \cdot K$, $K \cdot I$, $K \cdot K$, and so on, but that is about it. Definitions (14) and (15) represent the simplest possible choices; the choice $X = K$ is viable too, and you will discover fairly quickly that $F = I$ is not a good idea, in view of requirement (13) for F.

□

2.2 Why p is irrational, for every prime p

Here is a another non-trivial example. We wish to prove that $\sqrt{p}$ is *irrational*, for a given and, for the time being fixed, prime number p. In this section variables x and y range over the positive natural numbers.

To reach our goal, we introduce a function f with the following informal interpretation, for all (positive natural) x:

$$f{\cdot}x \;=\; \text{“the number of times } x \text{ is divisible by } p\text{”} \quad .$$

For our purpose we do not need a complete definition of f; all we need are these two properties, for our (fixed) prime p and for all x, y:

$$(16) \quad f{\cdot}p \;=\; 1$$
$$(17) \quad f{\cdot}(x*y) \;=\; f{\cdot}x + f{\cdot}y$$

The word “irrational” meaning “not rational”, we must prove that $\sqrt{p}$ is not a rational number:

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

from which we conclude that $\sqrt{p}$ is irrational. In this proof we have not needed to assume that x and y have no common divisors – in contrast to other renderings of the “same” proof –.

* * *

In the above proof I seem not to have used anywhere that p is prime: so, the theorem is also valid for all other numbers? Of course not: that p is prime plays a role in the definition of f. In particular, property (17) is only valid for prime numbers p. In this way, calculational reasoning contributes to – and also is only possible with! – a good modularization of the proof.

The crux of the above proof is the property: $(\forall x, y :: 1{+}2*y \neq 2*x\,)$, and this immediately points to a generalization, because it is an instance of the following, more general property:

$$(\forall x, y :: r + n*y \neq n*x\,) \quad , \text{ for all } \; r, n : \; 0{<}r{<}n \quad .$$

Now it is no big step anymore – we leave this as an exercise – to construct the following, more general theorem, where p now ranges over all primes and f_p denotes the function, as above, associated with prime p:

For all positive natural n and z:

"$\sqrt[n]{z}$ is rational" $\Leftrightarrow$ $(\forall p :: f_p \cdot z \bmod n = 0)$.

The theorem is an *equivalence* now: it states the exact – necessary and sufficient – condition for rationality of any root. To *apply* this theorem and to evaluate this condition we need a full definition of f, but to *prove* the theorem we only need (16) and (17): that is Separation of Concerns too.

3 Generalization by Abstraction and other Techniques

3.0 Where imperative programming fails

Consider the problem how to compute X^N efficiently, for some given integer X and some given natural N.

In an imperative style of programming, we would start with formulating the required pre- and postconditions for this problem. The precondition simply is

$$0 \leq N \ ,$$

whereas the postcondition becomes, after having introduced a variable z, say, that will hold the answer:

$$z = X^N \ .$$

By replacing, in this postcondition, the constant N by a variable n, say, we propose as invariants for the inevitable repetition:

P0: $0 \leq n \leq N$

P1: $z = X^n$

Using these invariants the following little program can now be constructed easily:

program 0:

$z, n := 1, 0$
; do $n \neq N \ \rightarrow \ z, n := z * X \, , \, n+1$
od
$\{ \ z = X^N \ \}$

Notice that this program has linear time complexity.

As a matter of fact, this program is based on the following (recurrence) relations for exponentiation, for all natural i:

(18) $X^0 \ = 1$

(19) $X^{i+1} = X^i * X$

* * *

So far, so good, but exponentiation has other interesting properties, like, for all natural i:

(20) $X^{2*i} = (X^i)^2$, and:

(21) $X^{2*i} = (X^2)^i$.

These relations suggest that more efficient solutions, with logarithmic time complexity, should be possible, but with the above approach it is practically impossible to exploit this.

For quite some time we do know, however, that the use of a so-called *tail invariant* brings relief:

P0 : $0 \leq n \leq N$

P2 : $X^N = z * x^n$

Notice the differences between P2 here and P1 above: firstly, an additional variable, x, has been introduced, more about which later; secondly, in P1 variable z may be viewed as representing the "answer under construction", whereas in P2 variable z may be viewed as representing "what is missing from the answer". This difference is clarified somewhat better if we rewrite P1 , thus:

P2 : $X^N = z * X^{N-n}$

Using the new invariants we can now construct the following program, which indeed has logarithmic time complexity:

program 1 :

```
    x, z, n := X, 1, N
;   do  n ≠ 0  →  if  n mod 2 = 0  →  x, n := x * x , n div 2
                  []  n mod 2 = 1  →  z, n := z * x , n − 1
                  fi
    od
```

* * *

Apparently, a programmmer has to decide, in quite an early stage of his design, whether to obtain a tentative invariant by means of the simple technique of replacing a constant by a variable, or to use a tail invariant of some sort. The latter technique undoubtedly is more general but it is also more difficult, because there are so many possibilities. As a very simple example: both relation (20) and relation (21) seem, at first sight, equally viable; which is the one to be used?

The lesson to be learned here is that a programmer should investigate the relations between the values to be computed *first*, that is, *before* he chooses whatever invariant. In the case of exponentiation, for example, the decision to use P1 as an invariant must be considered premature. Here Functional Programming enters the picture.

3.1 Generalization by abstraction

In functional programming we solve the problem of exponentiation as follows. First, we introduce a function f , say, which is required to satisfy:

$f \cdot n = X^n$, for all $n : n \in \mathbb{N}$.

Two remarks are in order here: first, because this proposition is a requirement, a goal, we call it function f's *specification*. Mathematically speaking, a specification has the same status as a *theorem*: it must be proved. Second, this specification has been obtained by the very simplest form of generalization, namely by replacing constant N by variable n. Thus, the constant expression X^N becomes X^n, which is a function of n.

I always have considered it good programming practice to *verify immediately* that a proposed generalization serves its goal, that is, that the problem at hand indeed is an instance of it. In our example, this is trivial: the generalization has been obtained by replacing a constant by a variable, so by substituting that constant back for the variable we obtain our original problem back. So, X^N can be written as an application of f:

$$X^N \;=\; f{\cdot}N \quad .$$

By using properties (18) and (19) of exponentiation we can derive the following recursive definition for f. As a matter of fact, there is not much to derive here, as it mainly boils down to encoding these properties into the functional-programming language:

program 2:

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

Now, however, we can also exploit the other properties of exponentiation; using (20), for example, we derive, for $n : 1 \leq n$:

$$\begin{array}{ll} & f{\cdot}(2*n) \\ = & \quad \{ \text{ specification of } f \text{ } \} \\ & X^{2*n} \\ = & \quad \{ \text{ property (20) } \} \\ & (X^n)^2 \\ = & \quad \{ \text{ specification of } f\text{, by Induction Hypothesis } \} \\ & (f{\cdot}n)^2 \quad . \end{array}$$

Because this derivation only covers the case for even (and positive) arguments, we also need an alternative for odd arguments; for this we can use the same relation as in program 2. This can be encoded as a functional program as follows. Notice that the local name y for $f{\cdot}(n{+}1)$ is indispensable here: without it we should have written $f{\cdot}(n{+}1) * f{\cdot}(n{+}1)$, which would have annulled the gain in efficiency.

program 3:

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

In the derivation of this functional program we have used property (20); using property (21) instead we derive, again for $n : 1 \leq n$:

$$\begin{array}{ll} & f{\cdot}(2*n) \\ = & \quad \{ \text{ specification of } f \text{ } \} \end{array}$$

$$
\begin{array}{ll}
 & X^{2*n} \\
= & \quad \{ \text{ property } (21) \ \} \\
 & (X^2)^n \quad .
\end{array}
$$

Here we are stuck, however, because the expression $(X^2)^n$ is not an instance of the expression in f's specification. It does *resemble* f's specification, though: it actually is an instance of it, provided that we substitute X^2 for X. So, we conclude that we must also replace the constant X by a variable x, say. Thus, we obtain we new function g, say, with two parameters x and n, and with this specification:

$$g{\cdot}x{\cdot}n \;=\; x^n \quad \text{, for all } x : x{\in}\mathbb{Z} \text{ and } n : n{\in}\mathbb{N} \quad .$$

This is a very simple example of the principle of Generalization by Abstraction: we have expression X^n and, after some investigation, also $(X^2)^n$; these two expressions have in common that both are instances of the more general expression x^n.

In accordance with our "good programming practice" we verify again that the original problem indeed is an instance of this generalization. In this case this is trivial again, because the new parameter x also has been introduced by replacing a constant by a variable. Therefore, we have:

$$f{\cdot}n \;=\; g{\cdot}X{\cdot}n \quad \text{, for all } n : n{\in}\mathbb{N} \quad \text{, and, hence, as a special case:}$$

$$X^N \;=\; g{\cdot}X{\cdot}N \quad .$$

If we generalize a problem then all work that already has been done on the special instance must, at least in principle, be redone: the work on the original special case looses its validity for the more general case. In practice, however, the situation very often is not that bad. If the work already done does not depend on properties that are valid only for that special case, then that work *may retain* its validity. In our exponentiation example, the above derivation that got stuck in the expression $(X^2)^n$ does not depend on particulary properties of X. As a matter of fact, X already is a (meta-)parameter of the problem to start with! Therefore, all recurrence relations about exponentiation, in the form of properties (18) through (21), remain valid when X is replaced by x: these properties hold for any X.

So, after having replaced "constant" X by parameter x we can now complete that derivation, for our new, more general, function g; actually, here we do redo that derivation because it is so short, again for $n : 1 \leq n$:

$$
\begin{array}{ll}
 & g{\cdot}x{\cdot}(2*n) \\
= & \quad \{ \text{ specification of } g \ \} \\
 & x^{2*n} \\
= & \quad \{ \text{ property } (21) \text{, with } x \text{ for } X \ \} \\
 & (x^2)^n \\
= & \quad \{ \text{ specification of } g \text{, by Induction Hypothesis (using } 1{\leq}n) \ \} \\
 & g{\cdot}(x*x){\cdot}n \quad .
\end{array}
$$

Together, these considerations give rise to the following functional program for g:

program 4 :

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

3.2 Accumulating parameters

A technique that is rather well-known by functional programmers lives under the name *accumulating parameters*. This is, however, nothing but an application of Generalization by Abstraction. In the second line of program 4 , for example, we have the expression $g{\cdot}x{\cdot}(2{*}n){*}x$. Because multiplication has 1 as its identity element we can also rewrite $g{\cdot}x{\cdot}n$ equivalently as $g{\cdot}x{\cdot}n{*}1$; now both these expressions are instances of the more general pattern $g{\cdot}x{\cdot}n{*}y$, where y is a fresh variable.

Thus, we obtain a further generalization of our problem. Calling the new function h , its specification then becomes:

$$h{\cdot}y{\cdot}x{\cdot}n \;=\; x^n{*}y \quad \text{, for all } y,x : x{\in}\mathbb{Z} \text{ and } n : n{\in}\mathbb{N} \quad .$$

The additional parameter y often is called an "accumulating parameter" because, as we will see, the answer of the computation is accumulated in it. The above shows, however, that this operational connotation is not needed to introduce such a parameter: this is just another application of Generalization by Abstraction.

Completing this development now is rather straightforward. Firstly, we have already observed that $g{\cdot}x{\cdot}n \;=\; g{\cdot}x{\cdot}n{*}1$; hence, we conclude that:

$$g{\cdot}x{\cdot}n \;\;=\;\; h{\cdot}1{\cdot}x{\cdot}n \quad ,$$

and deriving a recursive definition for h is not difficult either, yielding this program:

program 5 :

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

In the base case we now have $h{\cdot}y{\cdot}x{\cdot}0 \;=\; y$: indeed, the function's "final" value is the value "accumulated" in y . (But: so what?)

The definition of h has the special property that it is *tail-recursive*. Tail-recursive definitions of functions directly correspond to iterative sequential programs. For example, if we transform program 5 into an iterative sequential program we obtain our earlier program 1 almost literally, save a few minor differences. This also clarifies now where, in program 1, the additional variable x comes from: it has emerged in our first generalization, from function f to g .

Finally, note that the expression we started with can be expressed in terms of h by:

$$X^N \;=\; h{\cdot}1{\cdot}X{\cdot}N \quad .$$

3.3 Tupling

If two, or more, functions on the same domain have definitions that exhibit the *same patterns of recursion*, then these definitions can be *paired*, or more generally *tupled*. That

is, we replace the pair, or tuple, of functions by a single function with pairs, or tuples, for its value.

More precisely, if we have a function f, of some type $X \to V$, and a function g, of some type $X \to W$, then their pairing yields a function h, say, of type $X \to V \times W$, and with this specification:

$$h \cdot x = \langle\, f \cdot x \,,\, g \cdot x \,\rangle \quad \text{, for all } x : x \in X \quad .$$

As a matter of fact, this can be considered as a kind of generalization too. If f is the function of our interest, and if g only has been introduced for the sake of f, then the transition from f to h is a generalization: the range of function f, which is V, is *extended* so as to include the range, W, of g as well.

Function h is more general than f indeed, because now f can be defined in terms of our new function h:

$$f \cdot x = v \quad \mathsf{whr}\ \langle\, v \,,\, w \,\rangle = h \cdot x\ \mathsf{end} \quad .$$

This is a meaningful transformation provided that we are able to construct a definition for h in which f and g do not occur anymore. Well, if the patterns of recursion in the definitions of f and g are sufficiently similar, the definition for h will exhibit that same pattern of recursion, and its construction usually is a walk-over.

Tupling is an important technique because it can give rise to drastic improvements in the efficiency of a program: from quadratic to linear, or even from exponential to linear. Such improvements are due to – operationally speaking – the *synchronization* of the computations for $f \cdot x$ and $g \cdot x$.

* * *

To illustrate how tupling works we consider the following definition of the well-known Fibonacci numbers, for natural n:

program 6 :

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

Although this is a correct functional program it is rather inefficient, because the evalution of $\mathit{fib} \cdot n$ will require an amount of time that is exponential in n.

By means of tupling the efficiency can be improved drastically, but for tupling we need at least one additional function: where is that function? The answer is that the subexpression $\mathit{fib} \cdot (n{+}1)$ is a function of n as well; all we have to do is to name it and isolate it. Calling this function gib, it is a fairly straightforward exercise – using $\mathit{gib} \cdot n = \mathit{fib} \cdot (n{+}1)$ as gib 's specification – to rewrite fib 's definition as follows:

program 7 :

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

Now we have two functions on the same domain, with definitions of the same structure: both have a base case for 0, and both function values for $n{+}1$ are defined recursively in terms of $fib{\cdot}n$ and $gib{\cdot}n$.

Introducing a function h for the pairing of fib and gib, its specification simply is:

(22) $\quad h{\cdot}n = \langle\, fib{\cdot}n\,,\, gib{\cdot}n \,\rangle$, for all $n : n{\in}\mathbb{N}$.

Conversely, fib can now be defined in terms of h by:

$$fib{\cdot}n \;=\; x \;\; \textsf{whr}\; \langle\, x\,,\, y \,\rangle = h{\cdot}n \;\; \textsf{end} \quad .$$

To obtain a definition for h we derive:

$$\begin{array}{ll}
 & h{\cdot}0 \\
= & \quad \{\text{ specification } (22)\text{, of } h \,\} \\
 & \langle\, fib{\cdot}0\,,\, gib{\cdot}0 \,\rangle \\
= & \quad \{\text{ definition of } fib \text{ and } gib\text{, from program 7 } \} \\
 & \langle\, 0\,,1 \,\rangle \quad ,
\end{array}$$

and, for natural n:

$$\begin{array}{ll}
 & h{\cdot}(n{+}1) \\
= & \quad \{\text{ specification } (22)\text{, of } h \,\} \\
 & \langle\, fib{\cdot}(n{+}1)\,,\, gib{\cdot}(n{+}1) \,\rangle \\
= & \quad \{\text{ definition of } fib \text{ and } gib\text{, from program 7 } \} \\
 & \langle\, gib{\cdot}n\,,\, gib{\cdot}n{+}fib{\cdot}n \,\rangle \\
= & \quad \{\text{ extract } fib{\cdot}n \text{ and } gib{\cdot}n\text{, to prepare for the next step } \} \\
 & \langle\, y\,,\, y{+}x \,\rangle \;\; \textsf{whr}\; x = fib{\cdot}n \;\&\; y = gib{\cdot}n \;\; \textsf{end} \\
= & \quad \{\text{ combine } x \text{ and } y \text{ into one single pair } \} \\
 & \langle\, y\,,\, y{+}x \,\rangle \;\; \textsf{whr}\; \langle\, x\,,\, y \,\rangle = \langle\, fib{\cdot}n\,,\, gib{\cdot}n \,\rangle \;\; \textsf{end} \\
= & \quad \{\text{ specification } (22)\text{, of } h\text{, by Induction Hypothesis } \} \\
 & \langle\, y\,,\, y{+}x \,\rangle \;\; \textsf{whr}\; \langle x, y \rangle = h{\cdot}n \;\; \textsf{end} \quad .
\end{array}$$

This derivation will always follow the same pattern: it can almost be carried out in a completely mechanical way. As a result we obtain as recursive definition for h:

program 8 :

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

3.4 Fibonacci by generalization

On the one hand, we have:

$$fib{\cdot}(n{+}2) \;=\; fib{\cdot}(n{+}1) \,+\, fib{\cdot}n \quad ,$$

which can be written as:

$$fib{\cdot}(n{+}2) \;=\; 1 * fib{\cdot}(n{+}1) \,+\, fib{\cdot}n \quad .$$

On the other hand, we can write:

$$fib \cdot n \;=\; 0 * fib \cdot (n{+}1) \;+\; fib \cdot n \quad .$$

Therefore, let us investigate the general pattern, for some fresh variable y:

$$y * fib \cdot (n{+}1) \;+\; fib \cdot n \quad .$$

By trying to derive a recursive definition for this expression, we will discover soon that a further generalization is indicated. That is, we must investigate the even more general pattern, for fresh variables x and y:

$$y * fib \cdot (n{+}1) \;+\; x * fib \cdot n \quad .$$

So we introduce a new function F, say, with this specification:

$$F \cdot x \cdot y \cdot n \;=\; y * fib \cdot (n{+}1) \;+\; x * fib \cdot n \quad \text{, for all } x, y, n \in \mathbb{N} \quad .$$

This is a generalization indeed because function fib can now be defined in terms of F, as follows:

$$fib \cdot n \;=\; F \cdot 1 \cdot 0 \cdot n \quad .$$

By means of rather straightforward calculations the following definition for F can be derived:

program 9 :

$$\begin{array}{lll} & F \cdot x \cdot y \cdot 0 & = \; y \\ \& & F \cdot x \cdot y \cdot (n{+}1) & = \; F \cdot y \cdot (y{+}x) \cdot n \end{array}$$

This program has linear time complexity, and it is tail-recursive.

3.5 An example with lists

A (finite) list is either *empty*, denoted by $[\,]$, or *composite*, obtained from an element and a list. The composition operator is denoted by $\triangleright$ ("cons"): the list obtained by adding element b to list s is denoted by $b \triangleright s$.

If all elements of a list have the same type B, say, then we speak of *lists over* B. That is, $[\,]$ is a list over B, and if $b \in B$ and if s is a list over B, then so is $b \triangleright s$. Lists over the integers simply are called *integer lists*.

With *sum* and *len* for functions mapping integer lists to the sum of their elements and their lengths, respectively, we define a (real-valued) function *avg* mapping a non-empty integer list to the average value of its elements, for all non-empty integer lists s:

$$(23) \quad avg \cdot s \;=\; sum \cdot s \,/\, len \cdot s \quad .$$

This is an acceptable definition, provided we also construct definitions for *sum* and *len*, which is not difficult:

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

During evaluation of $avg \cdot s$ list s probably will be traversed twice, once for the evaluation of $sum \cdot s$ and once for $len \cdot s$. Although the time complexity remains linear, we may wish to improve the efficiency (somewhat) by seeing to it that the list is traversed only once.

* * *

One way to achieve our goal is by means of tupling. The two definitions of sum and len exhibit the same pattern of recursion; therefore, they can be easily paired. That is, we introduce a function h, say, with this specification, for all integer lists s:

$$h \cdot s \;=\; \langle\, sum \cdot s \,,\, len \cdot s \,\rangle \quad .$$

Function h is useful because now function avg can be redefined as follows:

$$avg \cdot s \;=\; x / y \;\; \mathsf{whr}\; \langle\, x, y \,\rangle = h \cdot s \;\; \mathsf{end}$$

By means of the standard tupling technique we then obtain as recursive definition for h:

program 10 :

$$\begin{array}{lll} & h \cdot [\,] & = \;\langle\, 0 \,,\, 0 \,\rangle \\ \& & h \cdot (b \triangleright s) & = \;\langle\, b{+}x \,,\, 1{+}y \,\rangle \;\; \mathsf{whr}\; \langle\, x \,,\, y \,\rangle = h \cdot s \;\; \mathsf{end} \end{array}$$

* * *

A second way to achieve our goal is to try and derive a definition for function avg directly, using (23) as its specification. For the empty list, however, avg is undefined, so we must take the one-element lists as the basis for the recursion. This then yields:

$$avg \cdot (b \triangleright [\,]) \;=\; b \quad .$$

For integer b and non-empty integer list s –so, $b \triangleright s$ is a list with at least two elements–, we now derive:

$$\begin{array}{ll} & avg \cdot (b \triangleright s) \\ = & \quad \{\; \text{specification (23), of } avg \;\} \\ & sum \cdot (b \triangleright s) \;/\; len \cdot (b \triangleright s) \\ = & \quad \{\; \text{definitions of } sum \text{ and } len \;\} \\ & (b + sum \cdot s) \;/\; (1 + len \cdot s) \quad , \end{array}$$

and here we get stuck, because there no simple way in which the expression obtained can be viewed as an instance of avg 's specification.

There is a way out, though: the original expression $sum \cdot s \,/\, len \cdot s$ equals $(0 + sum \cdot s) \;/\; (0 + len \cdot s)$. Therefore, this expression and $(b + sum \cdot s) \;/\; (1 + len \cdot s)$ are instances of the more general form $(x + sum \cdot s) \;/\; (y + len \cdot s)$, where x and y are fresh variables.

So, we introduce a new function $avgg$, say, with this specification, for all integer x, natural y, and integer lists s:

(24) $\quad avgg \cdot x \cdot y \cdot s \;=\; (x + sum \cdot s) \;/\; (y + len \cdot s) \quad .$

Notice that we do not require list s to be non-empty anymore. In order to avoid undefinedness due to division by zero, it now suffices that $y+len\cdot s \neq 0$, which for natural y amounts to $y\neq 0 \vee s\neq[\,]$. So, this latter condition now becomes the precondition in function *avgg* 's specification.

In terms of *avgg* our original function *avg* can now be defined easily:

$$avg\cdot s \;=\; avgg\cdot 0\cdot 0\cdot s \quad .$$

A recursive definition for *avgg*, in which the list is traversed only once, now is as follows; notice that this definition has the additional advantage of being tail-recursive:

program 11 :

$$\begin{array}{lll} & avgg\cdot x\cdot y\cdot[\,] & = \quad x\,/\,y \\ \& & avgg\cdot x\cdot y\cdot(b\triangleright s) & = \quad avgg\cdot(x+b)\cdot(y+1)\cdot s \end{array}$$

Remark: I found this example in a manuscript on functional programming, where its authors had failed to write down specification (24); as a result, they were not able to verify that $avgg\cdot 0\cdot 0\cdot x$ equals $avg\cdot x$. Not surprisingly, in their manuscript they did pay attention neither to program correctness, nor to specifications.
□

4 Misconceptions

4.0 Self-application

I must admit that, initially, it has taken several years before I came to grips with the kind of game I was playing. At the outset I approached the subject rather naively, discovering the rules of the game with ups and downs as I went along.

I remember, for example, that in those early years Kees Hemerik posed me annoying questions, like: "That is all very well, but what about self-application?" . At first I was not even aware that there might be a problem in the first place.

4.0.0 A paradox?

At a given moment, however, I decided that it was about time to study a text book on Denotational Semantics, preferably one that also covered the λ-calculus, because the λ-calculus is the model underlying functional programming. In the book by Joseph Stoy on this subject I encountered the following fragment.

quote: An even more fundamental difficulty, however, is highlighted by the fact that our definition [...] involves the application of [...] to itself. The possibility of self-application can lead to paradoxes. For example, suppose we define

$$u \;=\; \lambda y.\ \mathsf{if}\ y(y)=a\ \mathsf{then}\ b\ \mathsf{else}\ a \quad .$$

Then an attempt to evaluate $u(u)$ gives

$$u(u) \;=\; \mathsf{if}\ u(u)=a\ \mathsf{then}\ b\ \mathsf{else}\ a \quad ,$$

which is a contradiction [under the additional assumption $a\neq b$ (RH)].
□

The alleged contradiction in this example, however, is not due to the occurrence of self-application, as the author suggests, but by his failing to be explicit about the rules of his game. As is often the case with paradoxes, the contradiction disappears when these rules are made explicit. In this example, the symbol " $=$ " in the subexpression " $y(y)=a$ " is the culprit: the contradiction follows from the author's tacit assumption that " $=$ " denotes equality; in the λ-calculus, which is what the author is talking about here, such a general-purpose equality test simply does not exist. Actually, the above would-be paradox *proves* this non-existence: if it would be possible to encode such a general-purpose equality test in the λ-calculus then we would have a contradiction indeed; hence, it does *not* exist! So, Joseph Stoy was being sloppy here and, as a result, he was erring.

4.0.1 The typing issue

That one particular example turns out to be invalid does not, all by itself, mean that there is no problem with self-application. So, let us investigate it a little further.

One of the other alleged problems with self-application has to do with the *types* of the objects involved. If we admit expressions of the form $f\cdot f$, then what is the type of f in such a formula?

The obvious answer seems to be: f occurs as the left-hand argument of a function application, so f should be a function, of some type $X\to V$, say, for types X and V yet to be determined. On the other hand, f also occurs as the right-hand argument of a function application, so f should be an element of that function's domain, which we have named X.

Thus, we conclude that, from the point of view of typing, $f\cdot f$ is meaningful if both $f\in X\to V$ and $f\in X$. Therefore, it would be nice if X and V would satisfy:

$$X\to V \subseteq X \quad ,$$

because then *every* function in $X\to V$ would also be an element of its own domain. Then for *every* function in $X\to V$ we would be allowed to write $f\cdot f$, which would be an element of V.

For nontrivial sets X, V this kite will not fly, however, because by Cantor's theorem we know that, except for some degenerate cases, $X\to V$ has strictly greater cardinality than X. So, the required set inclusion is impossible and, hence, the obvious answer will not do.

The only other simple possibility – apart from rejecting self application altogether – is to require that X and V satisfy:

(25) $\quad X \subseteq X\to V \quad .$

This does not make $f\cdot f$ meaningful for every f in $X\to V$, but it does so for all f in X. As we shall show this is more than sufficient and, by the discrepancy between the cardinalities of $X\to V$ and X, this is about the best we may expect to obtain.

Nevertheless, X is a strange set. Its elements also are elements of $X\to V$, so they are functions, but they also are the arguments to which these functions can be applied. Do such sets exist? Lex Bijlsma has shown that in the standard set-theoretical model the answer is: no!

The only way out, therefore, is not to answer this question, but to sail around it, by taking the set inclusion in formula (25) with a grain of salt, that is, modulo a change of representation. By the way, this also is the course taken by all other "models" that pretend to solve this problem.

4.0.2 Resolving the issue

In order to decide what is the proper view on self-application, we do not need the λ-calculus. As a matter of fact, the problem can be discussed in terms of an extremely simple model, of which the λ-calculus is just a special case.

As in Section 2.0, we consider a set Ω on which we have a binary infix operator, which we write as "$\odot$" ("dottle") for this occasion, so as to avoid any possible confusion with the "$\cdot$" used for ordinary function application. So, $\odot$ has type $\Omega\times\Omega\to\Omega$.

remark: You may interpret operator $\odot$ as "function application", if you like, but for the sake of the following discussion this is irrelevant: it is just an abstract operator, and the whole discussion is independent of whatever connotations you may have in mind for it. But, eventually, of course, $\odot$ will be function application.
□

Now let f be any element in Ω, so $f\in\Omega$. Using this f we now define a function F, say, of type $\Omega\to\Omega$, as follows:

$$F\cdot x \;=\; f\odot x \quad \text{, for all } x : x\in\Omega \quad .$$

This function F depends, of course, on f: for *every* $f\in\Omega$ a corresponding function F exists. Thus, we can say that every $f\in\Omega$ *represents* a function in $\Omega\to\Omega$.

To make this representation explicit we introduce a function φ, of type $\Omega\to(\Omega\to\Omega)$. Function φ is the (so-called) *abstraction function* that maps elements of Ω –the concrete datatype– to functions in $\Omega\to\Omega$ –the abstract datatype–.

The relation between elements and functions is captured formally by the following definition of φ:

$$(\varphi\cdot f)\cdot x \;=\; f\odot x \quad \text{, for all } f,x\in\Omega \quad .$$

Notice that $\varphi\cdot f$ is the function represented by f and that, hence, $(\varphi\cdot f)\cdot x$ is the value of that function for argument x. By its definition this value equals $f\odot x$. But this means that the expression $f\odot x$ equals the application of "the function represented by f" to x. In this representation, operator $\odot$ now represents –or *implements*, if you like– function application!

Now we know how to interpret expressions like $f\odot f$: it is *not* the application of a function to itself, it is the application of "the function represented by f" to f. Put differently, looking from the abstract side: it is the application of a function to that function's representation. Nothing is wrong with that!

As a result, we do not have, as in formula (25):

$$\Omega\subseteq\Omega\to\Omega \quad ,$$

but we do have:

$$\varphi(\Omega)\subseteq\Omega\to\Omega \quad .$$

That is what I meant with "modulo a change of representation" at the end of Section 4.0.1.

When it is the operator $\odot$'s sole purpose to represent function application we may safely use $\cdot$ instead of $\odot$, thus (deliberately) ignoring the distinction between functions and their representations. Such an "abuse" of notation is quite common, it is harmless, and it simplifies the formulae; yet, the distinction is crucial to a proper understanding of self-application.

4.0.3 Conclusion

From the above only one conclusion can be drawn: self-application is *non-problem*.

If you feel tempted to consider my solution as cheating, you must keep in mind that the same representation trick is used in all other solutions to this problem.

4.1 Specifications

A common misconception in the functional programming community is that, in functional programming, we would not need a separate formalism for *specifications* because, as folklore has it, one would specify a problem by writing an "obviously correct" (but maybe very inefficient) functional program for it. Such an obviously correct program then is considered as an *executable specification* of the problem. If this program is efficient enough as well, the problem is considered solved; otherwise, the program has to be converted into an efficient one by means of systematic program transformations.

In this view, programming amounts to writing obviously correct programs and performing program transformations. Although convincing examples of the effectiveness of this approach do exist, this view is too limited, though, and it is so for at least three reasons.

Firstly, a specification is the first formalization of a problem, and, therefore, there is no such thing as the *correctness* of a specification: that a specification really captures what was (informally) intended cannot be *proved* but can only be *validated* by interpretation. To make such validation as reliable as possible it is important that the formalism used is as least restrictive as possible: a programming language, by virtue of its executability, is more restrictive than the whole of the mathematical language. (Apart from this, constructing specifications and rapid prototyping are two entirely different activities!)

Sometimes, however, writing down that trivially correct program is not a trivial task at all, whereas writing down a specification often is much easier, particularly so if the specification assumes the shape of an (implicit) equation for which the program has to yield a solution; typical examples are sorting, number conversion, and compilation.

Secondly, a specification is a formal representation of the required properties of a program, and (preferably) of *nothing more*. Avoiding *overspecification* is difficult enough in itself, but executable specifications always are overspecific, because apart from specifying a solution they also embody an algorithm to compute it. It is much more difficult to derive, say, QuickSort by means of program transformations from, say, Insertion Sort than to derive it from a more neutral specification.

Thirdly, a specification is an *interface*, namely between the *definition* (or, if you like, *internal structure*) of an object and its *use* (or *external properties*). This is important because an object's internal structure often is more complicated than its external properties: generally, a specification is simpler than an implementation.

In other words, a specification is a *firewall* between the use of a mechanism and its construction: as long as its specification remains the same, a mechanism's construction may be changed without affecting its use, and vice versa.

Similarly, a specification is for a programmer what a theorem is for a mathematician: a theorem is the interface between its proof and its use. A carpenter who constructs a right angle by forming a triangle from slats of lengths of 3, 4, and 5 feet does not need to know how Pythagoras's theorem is proved: all he has to rely on is the existence of such a proof.

4.1.0 The maximal segment-sum

A *segment* of a finite list s is any finite list "occurring (consecutively) somewhere in" s. Formally, this means that a segment of s is any finite list y for which finite lists x and z exist satisfying:

$$x +\!\!+ y +\!\!+ z = s \quad .$$

A well-know programming problem is the problem of the computation of the maximal value of the sums of all segments of a given finite sequence of integers. As a functional program, this problem can be specified by introducing a function f, say, that maps an integer list to the maximum of the sums of all segments of that list.

Using the above characterization of segments, function f can now be concisely specified by, for all integer lists s:

$$f \cdot s \;=\; (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = s : sum \cdot y\,) \quad .$$

Notice the implicit nature of this specification: the segments of interest are represented by bound variable y and the set of values of this variable is given as the solutions of an equation.

* * *

For the sake of contrast, here is an executable specification for the same function f, in the form of an "obviously [?] correct" functional definition:

$$f \;=\; foldr \cdot (\mathsf{max}) \cdot 0 \circ (sum\,\bullet) \circ segs \quad , \tag{26}$$

where function *segs* maps a list to *a list* (actually representing *the set*) containing all segments of that list:

$$segs \;=\; foldr \cdot (+\!\!+) \cdot [\,] \circ (inits\,\bullet) \circ tails \quad . \tag{27}$$

To make this "specification" complete we must still supply definitions for the functions *inits* and *tails*, but we omit these here. (Their construction is not trivial, though.) To understand the validity of this "specification" we must also know enough properties of the (standard) functions *foldr*, *filter* and $(\bullet)$ ("map"). All this having been said and done, this "specification" is neither trivially validated nor easily constructed: it is way too complicated.

Formulae (26) and (27) are overspecific because they contain a few premature design decisions. Instead of *foldr*, for instance, we might have equally well used *foldl*, and in this stage of the development it is not at all clear which is the one to be preferred. Because *foldr* occurs twice we may not expect that the choice is entirely irrelevant. (This dilemma could and, hence, should have been avoided by using the neutral function *fold*, which is possible because both max and $+\!\!+$ are associative.)

Similarly, a segment can be defined as an initial segment of a tail segment of the list, but also as a tail segment of an initial segment of the list: instead of $(inits\,\bullet) \circ tails$ we might have equally well used $(tails\,\bullet) \circ inits$, and, again, in this stage it is difficult to foresee what the consequences of this choice will be. Nevertheless, this choice *must* be made because it cannot be circumvented. This is another example of overspecification.

4.1.1 Sorting

When formulated as a functional program, a sorting algorithm for integer lists is a function *sort*, say, that maps integer lists to integer lists, in such a way that the list that is the function's value satisfies two requirements:

- this list contains the same numbers, equally often, as the function's argument;
- this list is ascending.

The first requirement can be formalized by requiring that the function's value *represents* the same bag of integers as the function's argument; the second requirement can be formalized by requiring that the function's value satisfies a predicate that characterizes ascendingness.

So, the specification of a function *sort* as a sorting algorithm can be formulated concisely as follows, for all integer lists s, t:

$$sort \cdot s = t \;\Rightarrow\; \mathcal{B} \cdot t = \mathcal{B} \cdot s \wedge \mathcal{A} \cdot t \quad .$$

Here $\mathcal{B}$ denotes the abstraction function from integer lists to integer bags, and $\mathcal{A}$ denotes the predicate that is `true` if and only if its argument is ascending.

To make this specification complete, we must also formulate suitable definitions for both $\mathcal{B}$ and $\mathcal{A}$, of course, but these definitions do not have to be executable. Fortunately, these definitions can be kept simple too.

* * *

An executable "specification" for sorting probably will be something like:

$$sort \;=\; head \circ filter \cdot asc \circ perms \quad ,$$

where function *perms* maps a finite list to a list containing all permutations of its argument, and where $filter \cdot asc$ takes a list of integer lists as its arguments and filters out the ones for which boolean function *asc* yields `true`. Here function *asc* is required to be a (computable) implementation of our (executable or non-executable) predicate $\mathcal{A}$.

To make this specification complete, additional definitions for *filter* and *perms* must be constructed. Particularly, the latter obligation is a non-trivial affair: correctly enumerating all permutations of a list is a non-trivial exercise. Moreover, this obligation, again, suffers from serious risk of overspecification: actually, we are only interested in the *set* of all permutations, so enumerating them as a list is overspecific. And why? Because one is trying to force the specification, at all costs, into the straitjacket of a functional-programming language, which does have notations for lists but not for sets or bags? I don't think so!

Finally, transforming such a, terribly inefficient, specification into a decent and useful sorting program will be troublesome at best.

4.1.2 Conclusion

Until this very day, I have never ever encountered a text book on functional programming in which specifications were given the role they deserve. For example, I have never seen a text book giving a formal specification of a sorting program. Yet, I have seen text books containing functional definitions of sorting algorithms. Needless to say that these

definitions were not *designed* in any reasonably systematic way, nor was their correctness proved. They were just pulled out of the magician's hat: that is the only thing the author(s) of such a text can do, because without a proper specification there is nothing to be designed nor proved. The readers of such a text may be pleased to find such algorithms, but presenting algorithms in this way has nothing to do with programming as a constructive discipline.

5 Epilogue

It really is regrettable that programming as a mathematical activity in general, and, more specifically, correctness by design and calculational techniques, have disappeared from our curricula. Programming is intrinsically difficult, in whatever way it is practised. As I have tried to illustrate today, the above principles have proven their effectiveness, not only for programming but for mathematical reasoning in general, and they deserve to be remembered and to be developed further.

* * *

When, about a year ago, I told Jan Friso Groote that I would be retiring soon, his immediate response was: "Why, everything is going so well these days?"

He was right: everything has been going well, and over the years I have enjoyed my profession in general, and teaching in particular. But, it is better to quit right when everything still is going well, instead of letting fade away my capabilities and my commitment.

Therefore, after 29 uninterrupted years at this university, most of which I have spent with pleasure, I thank you for your attention and I wish to conclude with these words:

Het is mooi geweest zo, het is tijd voor iets anders . . .

Eindhoven, 26 august 2013

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven