Let's not make things worse

0 On 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 a "trivially correct" (and possibly very inefficient) functional program for it. Such a trivially 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 boils down to writing trivially correct programs and performing program transformations. 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 validation as easy 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 proper-*

ties)[0]. This is important because an object's internal structure often is more complicated than its external properties: generally, a specification is simpler than a definition.

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

$$avg{\cdot}x \;=\; sum{\cdot}x \,/\, len{\cdot}x \quad .$$

During evaluation of $avg{\cdot}x$ list x probably will be traversed twice, once for the evaluation of $sum{\cdot}x$ and once for $len{\cdot}x$. A more efficient definition in which the list is traversed only once is:

$$\begin{array}{lcl} avg{\cdot}x & = & avg1{\cdot}0{\cdot}0{\cdot}x \\ avg1{\cdot}s{\cdot}n{\cdot}[\,] & = & s \,/\, n \\ avg1{\cdot}s{\cdot}n{\cdot}(b \triangleright x) & = & avg1{\cdot}(s{+}b){\cdot}(n{+}1){\cdot}x \quad . \end{array}$$

To verify that $avg1{\cdot}0{\cdot}0{\cdot}x$ indeed equals $avg{\cdot}x$ is easier if we use as specification for *avg1*:

$$(0) \qquad (\forall s, n, x ::\; avg1{\cdot}s{\cdot}n{\cdot}x \;=\; (s{+}sum{\cdot}x) \,/\, (n{+}len{\cdot}x)\;) \quad ,$$

from which it immediately follows that: $avg1{\cdot}0{\cdot}0{\cdot}x \;=\; sum{\cdot}x \,/\, len{\cdot}x$. This equality now only depends on specification (0), of *avg1*, and not on its recursive definition, which is only needed to verify, once and for all, that *avg1* satisfies (0). In this way, separate specifications contribute to the modularisation of correctness proofs. Modularisation is important because it allows us to reconsider design decisions, without the obligation to reconsider the correctness of the whole design.
(I found this example in a manuscript on functional programming, where its authors failed to write down formula (0); as a a result, they were not able to verify that $avg1{\cdot}0{\cdot}0{\cdot}x$ equals $avg{\cdot}x$. Not surprisingly, in their manuscript they did not pay any attention to program correctness.)

□

[0] As in mathematics, where a *theorem* is the interface between its proof and its use.

1 An example

We are interested in a formal specification of and a definition for a function F, of type $\mathcal{L}_*(\mathsf{Int}) \to \mathsf{Nat}$, such that it satisfies the informal characterisation, for all integer lists s:

$$F{\cdot}s \;=\; \text{“the length of a longest } \textit{segment} \text{ of } s \text{ containing zeroes only”} \quad .$$

This characterisation has three main ingredients, namely the notion of a *segment* of a list, the predicate “containing zeroes only” and the notion of the *length* of a list. The latter two are most easily formalised: both are functions on lists. More precisely, the predicate is a boolean function Q (say) and the length is a natural function L (say). Because these functions admit several, slightly different but equivalent, definitions we shall not formulate these definitions until we need them. In addition, both the problem and its solution are to a large extent independent of the details of Q and L.

A list y is a segment of list s if (and only if) lists x and z exist satisfying:

$$x +\!\!+ y +\!\!+ z \;=\; s \quad ,$$

and y is a segment satisfying Q of list s if lists x and z exist satisfying:

$$x +\!\!+ y +\!\!+ z \;=\; s \;\wedge\; Q{\cdot}y \quad .$$

Our function F can now be specified thus:

$$F{\cdot}s \;=\; (\,\mathsf{max}\, x, y, z : x +\!\!+ y +\!\!+ z = s \wedge Q{\cdot}y :\; L{\cdot}y\,) \quad \text{, for all } \; s \quad . \tag{1}$$

The introduction of names Q and L not only enhances modularisation and clarity but also serves to keep the length of this formula within reasonable limits. If we were to substitute explicit formulae for $Q{\cdot}y$ and $L{\cdot}y$ the specification of F would become unmanageable and those parts of the following derivations that are independent of the properties of Q and L would become needlessly laborious.

intermezzo: For one reason or another, you may dislike formula (1), but it is the most concise specification for this problem I can think of. Moreover and more importantly, in this shape the specification is most easily *validated*, because formula (1) is a direct formal translation of the informal “the length of a longest segment of s containing zeroes only”.

For the sake of contrast, here is an executable specification in the form of an “trivially correct” functional definition:

(2) $F = foldr\cdot(\mathsf{max})\cdot(-\infty) \circ (L\bullet) \circ filter\cdot Q \circ segs$,

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

(3) $segs = foldr\cdot(+\!\!+)\cdot[\,] \circ (inits\bullet) \circ tails$.

To make this "specification" complete we must still supply definitions for the functions *inits* and *tails*, but we omit these here. 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 much too complicated.

Formulae (2) and (3) 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.

Finally, as the time complexity of (2) and (3), viewed as an executable program, is at best $\mathcal{O}(n^3)$, we are still left with the programming task of transforming this into a more efficient definition. In view of the complexity of this "specification" this task may be expected to be quite laborious.

□

We now derive a recursive definition for F by induction on (the structure of) s; we shall introduce the relevant properties of Q and L as we proceed:

$F\cdot[\,]$

$=$ { specification (1) of F }

$$
\begin{array}{ll}
 & (\,\mathsf{max}\,x,y,z : x +\!\!+ y +\!\!+ z = [\,] \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ x +\!\!+ y +\!\!+ z = [\,] \text{ has only one solution } \} \\
 & (\,\mathsf{max}\,x,y,z : x = [\,] \wedge y = [\,] \wedge z = [\,] \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ \text{one-point rule, } \bullet \text{ assuming } Q\cdot[\,]\ \} \\
 & L\cdot[\,]\ .
\end{array}
$$

Hence, we can define $F\cdot[\,]$ as $L\cdot[\,]$, provided predicate Q satisfies $Q\cdot[\,]$. In our case, Q is the predicate "containing zeroes only" and because the empty list certainly contains zeroes only, we may safely define:

(4) $\quad Q\cdot[\,] \ \equiv\ \mathsf{true}\ .$

Similarly, as L is the length function, we have:

(5) $\quad L\cdot[\,] \ =\ 0\ ,$

so we can define:

(6) $\quad F\cdot[\,] \ =\ 0\ .$

For non-empty lists of the shape $b \triangleright s$ (b "cons" s) we derive:

$$
\begin{array}{ll}
 & F\cdot(b \triangleright s) \\
= & \quad \{\ \text{specification } (1) \text{ of } F\ \} \\
 & (\,\mathsf{max}\,x,y,z : x +\!\!+ y +\!\!+ z = b \triangleright s \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ \text{range split } x = [\,] \vee x \neq [\,]\ \} \\
 & (\,\mathsf{max}\,y,z : y +\!\!+ z = b \triangleright s \wedge Q\cdot y :\ L\cdot y\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\,x,y,z : x \neq [\,] \wedge x +\!\!+ y +\!\!+ z = b \triangleright s \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ \text{introduction of a new function } G\ \} \\
 & G\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\,x,y,z : x \neq [\,] \wedge x +\!\!+ y +\!\!+ z = b \triangleright s \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ x \neq [\,] :\ \text{dummy transformation } x := b \triangleright x\ \} \\
 & G\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\,x,y,z : (b \triangleright x) +\!\!+ y +\!\!+ z = b \triangleright s \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ \text{list properties (simplification)}\ \} \\
 & G\cdot(b \triangleright s) \quad \mathsf{max} \quad (\,\mathsf{max}\,x,y,z : x +\!\!+ y +\!\!+ z = s \wedge Q\cdot y :\ L\cdot y\,) \\
= & \quad \{\ \text{specification of } F\text{, by Induction Hypothesis}\ \} \\
 & G\cdot(b \triangleright s) \quad \mathsf{max} \quad F\cdot s\ .
\end{array}
$$

Thus, for non-empty lists, F can be defined recursively by:

(7) $\quad F{\cdot}(b \triangleright s) \;=\; G{\cdot}(b \triangleright s) \;\mathsf{max}\; F{\cdot}s \quad .$

In the last derivation we have needed no further properties of Q or L, because we have only manipulated dummy x which does not occur in applications of Q or L. An experienced functional programmer probably will shorten the derivation by performing the range split and the subsequent dummy transformation in a single step. This particular dummy transformation is justified by the property that $\triangleright$ ("cons") is *injective*, that is:

$$b \triangleright x \;=\; c \triangleright y \quad \Rightarrow \quad b = c \wedge x = y \quad .$$

The introduction of a *new* function G is justified by the emergence of a *new* formula that is not an instance of the formula for which we are deriving a definition. Function G has the same type as F and its specification is:

(8) $\quad G{\cdot}s \;=\; (\,\mathsf{max}\, y, z : y +\!\!+ z = s \wedge Q{\cdot}y :\; L{\cdot}y\,) \quad$, for all $\; s \quad .$

Informally, $G{\cdot}s$ is the length of a longest *inital segment* of s containing zeroes only. The specification of G is similar to but simpler than F's specification, so we have reduced the problem to a simpler problem. As a definition for $G{\cdot}[\,]$ we obtain, in very much the same way as we derived for $F{\cdot}[\,]$:

(9) $\quad G{\cdot}[\,] \;=\; 0 \quad .$

Furthermore, for non-empty lists of the shape $b \triangleright s$ we derive:

$$\begin{array}{ll}
 & G{\cdot}(b \triangleright s) \\
= & \quad \{\ \text{specification (8) of } G\ \} \\
 & (\,\mathsf{max}\, y, z : y +\!\!+ z = b \triangleright s \wedge Q{\cdot}y :\; L{\cdot}y\,) \\
= & \quad \{\ \text{range split } y = [\,] \vee y \neq [\,] \text{, combined with } y := b \triangleright y\ \} \\
 & (\,\mathsf{max}\, z : z = b \triangleright s \wedge Q{\cdot}[\,] :\; L{\cdot}[\,]\,) \quad \mathsf{max} \\
 & (\,\mathsf{max}\, y, z : (b \triangleright y) +\!\!+ z = b \triangleright s \wedge Q{\cdot}(b \triangleright y) :\; L{\cdot}(b \triangleright y)\,) \\
= & \quad \{\ \text{one-point rule, using (4) and (5); simplification}\ \} \\
 & 0 \quad \mathsf{max} \quad (\,\mathsf{max}\, y, z : y +\!\!+ z = s \wedge Q{\cdot}(b \triangleright y) :\; L{\cdot}(b \triangleright y)\,) \\
= & \quad \{\ \bullet \text{ assuming (10) and (11)}\ \} \\
 & 0 \quad \mathsf{max} \quad (\,\mathsf{max}\, y, z : y +\!\!+ z = s \wedge b = 0 \wedge Q{\cdot}y :\; 1 + L{\cdot}y\,) \quad .
\end{array}$$

Here we have used two new properties of Q and L, namely:

(10) $Q\cdot(b\triangleright y) \quad\equiv\quad b=0 \;\wedge\; Q\cdot y$,

and:

(11) $L\cdot(b\triangleright y) \quad=\quad 1 + L\cdot y$.

Together with (4) and (5) these properties completely define Q and L.

The last formula in the above derivation contains a term $b=0$ which is constant in the sense that it is independent of the dummies y and z. If $b=0$ is false the range of the quantification is empty and the whole formula collapes to 0, whereas if $b=0$ is true the term may be omitted from the quantification and we can continue the derivation:

$$
\begin{array}{ll}
 & 0 \;\; \mathsf{max} \;\; (\,\mathsf{max}\, y, z : y +\!\!+ z = s \;\wedge\; Q\cdot y :\; 1 + L\cdot y\,) \\
= & \quad \{\ (1+)\ \text{distributes over}\ \mathsf{max}\ \} \\
 & 0 \;\; \mathsf{max} \;\; (1 + (\,\mathsf{max}\, y, z : y +\!\!+ z = s \;\wedge\; Q\cdot y :\; L\cdot y)\,) \\
= & \quad \{\ \text{specification of } G\text{, by Induction Hypothesis}\ \} \\
 & 0 \;\; \mathsf{max} \;\; (1 + G\cdot s) \\
= & \quad \{\ 0 \le 1 + G\cdot s\ \} \\
 & 1 + G\cdot s \quad .
\end{array}
$$

The last step in this derivation depends on the property that G is a natural function (because length is a natural function). Thus, for non-empty lists G can be defined recursively by:

$$
\begin{array}{lcl}
(12) \quad G\cdot(b\triangleright s) & = & \mathsf{if} \;\; b\neq 0 \;\; \rightarrow \;\; 0 \\
 & & [\!] \;\; b=0 \;\; \rightarrow \;\; 1 + G\cdot s \\
 & & \mathsf{fi} \quad .
\end{array}
$$

In summary, by collecting definition fragments (6), (7), (9), and (12) we obtain the following definitions for F and G. By means of *tupling* these definitions can be merged into a single definition, with linear time complexity; we leave this as an exercise.

$$
\begin{array}{lcl}
F\cdot[\,] & = & 0 \;\; , \\
G\cdot[\,] & = & 0 \;\; , \\
 & & \\
F\cdot(b\triangleright s) & = & G\cdot(b\triangleright s) \;\; \mathsf{max} \;\; F\cdot s \;\; , \\
G\cdot(b\triangleright s) & = & \mathsf{if} \;\; b\neq 0 \;\; \rightarrow \;\; 0 \\
 & & [\!] \;\; b=0 \;\; \rightarrow \;\; 1 + G\cdot s \\
 & & \mathsf{fi} \quad .
\end{array}
$$

2 A variation

The segments of list s can also be defined by means of the take-and-drop calculus: $s\lfloor i$ (s "drop" i) is the tail segment of s, obtained by removing the initial segment of length i, and $s\lfloor i\lceil j$ (s "drop" i "take" j) is the initial segment of length j of $s\lfloor i$. In terms of take and drop, a formula like $x +\!\!+ y +\!\!+ z = s$ amounts to $x = s\lceil i$ and $y = s\lfloor i\lceil j$ and $z = s\lfloor i\lfloor j$ (for some i, j). Therefore, function F can also be specified by means of the take-and-drop calculus; this requires a reference to the length of list s but yields a specification that lends itself equally well for a calculational derivation of a recursive definition. For list s of length n this specification reads:

$$F{\cdot}s \;=\; (\,\mathsf{max}\, i, j : 0{\le}i{\le}i{+}j{\le}n \,\wedge\, Q{\cdot}(s\lfloor i\lceil j) : j\,) \quad .$$

Here I have used that L is the length function and that j is the length of the segment involved, that is, I have used $L{\cdot}(s\lfloor i\lceil j) = j$. For other functions than length we must, of course, retain the full expression $L{\cdot}(s\lfloor i\lceil j)$.

If an irrelevant choice between two alternatives really is unavoidable then the pain of overspecification is lessened when the two alternatives are connected by a simple calculational rule. A segment of a list can be defined either as an initial segment of a tail segment of the list, or as a tail segment of an initial segment of the list. That both definitions yield the same notion of segments is rendered in the take-and-drop calculus by a very simple formula; this formula lessens the pain because it enables us to switch easily to the other alternative if this is necessary:

$$s\lfloor i\lceil j \;=\; s\lceil (i{+}j)\lfloor i \quad , \text{ for all } i, j :\, 0{\le}i{\le}i{+}j{\le}n \quad .$$

Eindhoven, 15 july 1998

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