# OO versus FP — a little case study

Maarten Fokkinga
*University of Twente, Dept Computer Science*
*PO Box 217, 7500 AE Enschede, The Netherlands*
*email:* fokkinga@cs.utwente.nl

Version of April 21, 1995

We describe what we think is the essence of Object-Orientedness (OO), and then show by means of an example how this style translates into a Functional Programming (FP) notation. The example is the case of two thermometers, with possibly different scales (Centigrades, Fahrenheit, or whatever), that can be used separately but also in a coupling in such a way that the coupled pair behaves consistently.

## Introduction

Object-orientedness (OO) is a buzzword nowadays. Many books, articles, courses, and what-have-you deal with the subject. However, quite some treatments are rather vague about the precise nature, let alone precise *definition*, of object-orientedness. This paper is one step in my own attempt to come to understanding with that notion, and in particular its relation to functional programming (FP). Efficiency is not my concern; ease of construction, correctness, maintenance and adaptability is what counts. I started off this investigation in the supposition that the FP model would be at least as good, if not superior, to the OO model in these respects. Having finished the task I'm no longer inclined to say so without reservation.

In the first part of this paper I try to give some precise definitions (though stated in natural language) about object-orientedness. I describe what I think is the essence of object-orientedness. In the second part I redo Henderson's [2] object-oriented modeling of 'coupled thermometers', and I show how it can be done quite faithfully in a purely functional notation. The example concerns modeling of two thermometers, with possibly different scales (Centigrades, Fahrenheit, or whatever), that can be used separately but also in a coupling in such a way that the coupled pair behaves consistently.

**Object-Orientedness** A program is *object-oriented* if: all or most of its variables and procedures are collected (put together, organised, structured) into so-called objects. An *object* is: a collection of variables and procedures (often called *methods*) that have exclusive access to the variables. For example, let object $A$ consist of variables $x, y, z$ and procedures $p, q$. Then nowhere outside the bodies of $p$ and $q$ these variables $x, y, z$ may be mentioned. Procedures $p, q$ of object $A$ are accessible from everywhere; $p$ may be denoted outside the object by something like $A.p$ and inside the object by something like self.$p$. The claim of the proponents of object-orientedness is that this notion of object may correspond closely to the objects in the real world problem area, and, consequently, there is a close correspondence between the program structure and the actions in the real world. Thanks to the close correspondence between program structure and real world, the program is easier to construct, maintain, adapt and so on.

A program object models something from the real world that has a state (that is, it has some attributes which may vary in time), and for which there are some distinguished methods through which the attributes may be updated, changed and inspected. In the area of person administration a clear example is a person; the state of a person may be its age and address. When dealing with geometry one may model a circle as an object; its state is its centre and radius, and there are methods to inspect these attributes and to change them according to, say, a translation or scaling. One can go a far end in modeling the world by objects. An extreme form of an object is a single number variable; its state is its current value, and its methods are arithmetic operations.

In order that a structuring into objects is practically feasible, the notion of class offers some indispensable help. A *class* is: a generic description of an object; it makes it possible to define many objects in a short way. For example, let class $C$ be defined as "(the object is to have) variables $x, y, z$ and procedures $p, q$". Now the declaration "let $A, B$ be objects of class $C$" defines two objects $A, B$, each of which has variables $x, y, z$ and procedures $p, q$. Having defined class $C$, there is no need any more to spell out the variables and procedures of each object separately. Semantically such a class is a class indeed; it is a set or collection of objects, and an object belongs to a class if it has at least the variables and procedures prescribed by the class.

In order to provide a further structure among the classes, the notions of subclass, superclass, and inheritance offer some essential help. For example, let class $C'$ be defined as "(the object is to have) all variables and procedures as prescribed by class $C$ and in addition also variables $u, v, w$ and procedure $r$". Then $C'$ is said to be a *subclass* of $C$, and $C$ a *superclass* of $C'$, and $C'$ is said to *inherit* the variables and procedures of $C$. Clearly, each object of the subclass $C'$ is also a member of the superclass $C$, and one class may have several subclasses and several superclasses. So, these notions allow new classes to be expressed in terms of other classes, thus saving duplication of code and offering further opportunities to model 'real world structure' in the program.

**OO languages versus OO thinking** An imperative programming language need not have special provisions for objects and classes, provided (as explained shortly) it is sufficiently expressive with respect to control of scope and extent. In a sufficiently versatile language one can express objects and classes and so on even if there are no special syntactic constructs for them. So object-orientedness is more a style of thinking, a manner of organising one's conceptual algorithm, than a new language feature. However, a language may offer some help and encouragement in adopting such a style by providing predefined syntactic constructs.

Let us now briefly explain the notions of control of scope and extent. *Scope control*, or control of visibility, is the ability to specify, for each name in a program, the regions of the program text from which the name can be referred to. Block structure provides for some elementary scope control; labeling with keywords like *private* or *public* may further increase the control. (Elsewhere I have shown that unrestricted lambda-abstraction gives a very high scope control [1].)

*Extent control*, or control of life-time, is the ability to specify, for each variable in the program, the duration during which storage is allocated for the variable. Again block structure provides some elementary extent control: storage for variables is allocated in a last-in first-out regime on the stack. Complete control is given to the programmer by means of the heap and its explicit allocation and deallocation (by *new* and *dispose* ).

Looking back at our description of object and class, you may convince yourself that the concepts are just specific ways of controlling the scope of some variable names. Deliberately we have said nothing about the duration of the storage allocation for those variables. It may be clear, however, that a simple last-in first-out regime will not suffice; use of the heap seems unavoidable. We conclude that syntactic features for objects and classes are only necessary if the language has not sufficiently expressive means for controlling the scope and extent. Yet, the notions of object and classes may be so helpful that they deserve their own syntactic construct (in the same way that a repetition has its own syntactic construct even though it can be expressed in terms of *goto* jumps.)

Functional programming languages have no specific provisions for denoting 'real world objects'. It is therefore interesting to see to what extent FP languages hamper or enable the thinking in terms of objects. We shall investigate this question in the case study below.

**Functional Programming** We assume familiarity with functional programming, and we discuss here only one way (the only way?) the objects in an OO program may be modeled in a functional program.

Let $A$ be an object with state $s$ and procedures (methods) $p, q$. So, $s$ is a collection of variables, and $p, q$ are the only way to access $s$, whether for inspection or changement. Let the semantics of $p$ be fully specified as follows: if $s = s'$ before the call of $p(x)$, then $f_p(x, s')$ is the result delivered by the call and $g_p(x, s')$ is the new state. Similarly for $q$.

In a functional program we choose to model object $A$ by a function $a$; function $a$ accepts a list $xs$ of requests (the successive calls of the form $p(x)$ and $q(y)$ ), and returns a list of responses (the results of the calls). The request $p(x)$ translates to an entry $P\,x$ in the request list; a request $q(y)$ translates to $Q\,y$. In order to keep track of the internal state $s$ of object $A$, we use an auxiliary function $a'$ that has an additional parameter representing the state:

$$a\ xs \quad = \quad a'\ initialState\ xs$$

where

$$\begin{array}{lcl} a'\ s\ (P\,x : xs) & = & f_p\ s\ x\ :\ a'\ (g_p\ s\ x)\ xs \\ a'\ s\ (Q\,y : xs) & = & f_q\ s\ y\ :\ a'\ (g_q\ s\ y)\ xs \quad . \end{array}$$

Actually, the above function definition corresponds to a class definition rather than one particular object; each *occurrence of (a call of)* function $a$ corresponds to one particular object $A$ of the class.

## A case study

Here we shall present Henderson's [2, Chapter 5] model of two coupled thermometers, and our FP model for the same thing. Each OO description and program text is followed by the corresponding FP description and text. We try to get as close a correspondence as possible; taking Henderson's final model as given. We improve upon Henderson's presentation in that we split up the development of the final text into separate steps; these are probably the same steps as taken by Henderson, but which he has not recorded or presented.

The next paragraph is quoted from Henderson [2, page 115–116]. It nicely demonstrates the context and relevance of the problem.

**The thermometer problem** "This is a very simple problem, but it is illustrative of a genre of problems to do with reactive user interface, where the user edits some area of the screen setting off a chain of events which then make changes to other areas of the screen. We imagine two thermometers, one giving readings in Fahrenheit and the other readings in Centigrade. We imagine a screen [. . . ] where the two thermometers are displayed graphically. We imagine that the user can edit (perhaps by pointing with a mouse) the reading on either thermometer. The consequence of this editing will be that both thermometers will change to show the new temperature, the appropriate conversion between the scales having been done automatically, behind the scenes. Thus, for example, if we click on 32 on the Fahrenheit thermometer we expect that thermometer to show that reading and the Centigrade thermometer to change to read 0. [. . . ]"

**An OO thermometer** A thermometer is a device with the following kinds of interaction with its environment:

- It receives a request to adjust its current internally stored temperature. Let us say that the request is of the form '$Set\,(n)$', where $n$ is the new temperature.
- It receives a request to display the current temperature. Let us say that the request is of the form '$Get$', and the output is a number.

Each thermometer uses just one fixed scale, and the user should conform to this consistently. For example, for a thermometer $T$ we would expect that:

$$a := T.Get;\ T.Set(37);\ b := T.Get\ldots \quad \approx \quad a := undef;\ b := 37\ldots \quad .$$

The implementation is straightforward:

```
class $Thermometer$ :
    var $temp$ initially $undef$.
    proc $Set\,(n) \;=\; temp := n.$
    proc $Get \;=\;$ return $temp.$
end    .
```

**An FP thermometer** A thermometer is a device with the following kinds of interaction with its environment:

- It receives a request to adjust its current internally stored temperature. Let us say that the request is of the form ‘ $Set\,n$ ’, where $n$ is the new temperature.
- It receives a request to display the current temperature. Let us say that the request is of the form ‘ $Get$ ’, and the output is a number.

Thus a thermometer transforms a list of requests to a list of responses. For example, for a thermometer $t$ we would expect that:

$$[Get,\ Set\,37,\ Get\ldots] \quad \overset{t}{\mapsto} \quad [undef,\ 37\ldots] \quad .$$

The implementation is straightforward:

```
$thermometer \;=\; f\,undef$
where
$f\,m\,(Set\,n\ :\ xs) \;=\; f\,n\,xs$
$f\,m\,(Get\ :\ xs) \;=\; m\ :\ f\,m\,xs \quad .$
```

**An OO Centigrade thermometer** A Centigrade thermometer interprets all data as degrees Centigrade, and is initially set to the temperature of freezing water. So, a Centigrade thermometer is but a specialisation of a generic one. Therefore we define $Centigrade$ as a subclass of $Thermometer$ . After the introduction of $C$ as an object of class $Centigrade$ , we expect:

$$a := C.Get;\ C.Set\,(37);\ b := C.Get\ldots \quad \approx \quad a := 0;\ b := 37\ldots \quad .$$

The subclass definition, for $Centigrade$ and $Fahrenheit$ , is straightforward:

```
class $Centigrade$ subclass of $Thermometer$ :
    initially $temp := 0.$
end
class $Fahrenheit$ subclass of $Thermometer$ :
    initially $temp := 32.$
end    .
```

**An FP Centigrade thermometer** A Centigrade thermometer interprets all data as degrees Centigrade, and is initially set to the temperature of freezing water. So, a Centigrade thermometer is but a specialisation of a generic one. After having defined *centigrade*, we expect:

$$[Get,\ Set\,37,\ Get\ldots] \quad \overset{centigrade}{\mapsto} \quad [0,\ 37\ldots] \quad .$$

The definition, for *centigrade* and *fahrenheit*, is straightforward:

$$\begin{array}{lcl} centigrade & = & thermometer \cdot (Set\,0\ :) \\ fahrenheit & = & thermometer \cdot (Set\,32\ :) \quad . \end{array}$$

**OO coupling** Let *Left*, *Right* be two thermometers, possibly with different scales (degrees Centigrade or Fahrenheit). Then the coupling of these is a device that also functions as one, single, thermometer. The difference with a proper thermometer is *only* in the form of the requests: these may be addressed both to *Left* (in the scale of the left one) and to *Right* (in the right scale). The observer knows that the outcome of a request to the left one is to be interpreted in the scale of the left one. For example, right after the introduction of $C$ and $F$ as objects of class *Centrigrade* and *Fahrenheit*, respectively, and their coupling, we expect:

$$a := F.Get;\ C.Set(100);\ b := F.Get\ldots \quad \approx \quad a := 32;\ b := 212\ldots \quad .$$

The coupling of thermometers $T, T'$ is done by $T.Couple(T');\ T'.Couple(T)$. So we conclude that *Couple* is yet another procedure of thermometers. In order to avoid duplication of code, the new procedure is to be specified in class *Thermometer* and not in the subclasses.

In order to implement the coupling Henderson takes the following approach. First, two new entities (var *companion* and proc *Couple*) are added to the class *Thermometer*; these keep track of the coupling, if any. Second, proc *Set* in the general thermometer class is adapted: it resets its companion, if present. Finally, in order to express *Reset* in a general way (without having to decide what kind of thermometer the companion is: Centigrade or Fahrenheit or whatever), two new procs are added to the subclasses: *SetK* and *GetK*. These procedures perform the conversion between the internal scale and a fixed scale, say Kelvin; they are meant to be local, auxiliary procedures, not to be used outside the class. Functions *k2c*, *c2k* perform the conversion from Kelvin to Centigrade and conversely; these are not elaborated here. By means of *SetK* and *GetK*, procedure *Reset* is now easily expressed. Thus we get:

```
class $Thermometer$ :
    var $temp$ initially $undef$, $companion$ initially $none$.
    proc $Couple\,(T)\ =\ companion := T$.
    proc $Set\,(n)\ =$
```

```
            $temp := n$; if $companion \neq none$ then $companion.Reset$ endif.
        proc $Get$ = return $temp$.
        proc $Reset$ = self.$SetK\,(companion.GetK)$.
    end
    class $Centigrade$ subclass of $Thermometer$ :
        proc $SetK\,(n)$ = self.$Set(k2c(n))$.
        proc $GetK$ = return $c2k\,$(self.$Get$).
        initially $temp := 0$.
    end
    class $Fahrenheit$ subclass of $Thermometer$ :
        proc $SetK\,(n)$ = self.$Set(k2f(n))$.
        proc $GetK$ = return $f2k\,$(self.$Get$).
        initially $temp := 32$.
    end    .
```

In order to type check these definitions, and in particular procedure *Reset*, it might be needed to announce procedures *SetK* and *GetK* already in the superclass *Thermometer*.

**FP coupling** Let $left, right$ be two thermometers, possibly with different scales (degrees Centigrade or Fahrenheit). Then the coupling $couple(left, right)$ is a device that also functions as one, single, thermometer. The difference with a proper thermometer is *only* in the form of the requests. Each request to a couple is to be labeled with $L$ (the request is meant for *left* in the scale of *left*) or with $R$ (meant for *right* in the right scale). The observer knows that the outcome of a request to the left one is to be interpreted in the scale of the left one. For example, when $c = centigrade$ and $f = fahrenheit$ we expect:

$$[R\,Get,\ L(Set\,100),\ R\,Get,\ \ldots] \quad \overset{couple(c,f)}{\mapsto} \quad [32,\ 212, \ldots] \quad .$$

In order to implement the coupling we deliberately take the same approach as in the OO case: a *Set* to *left* has as “side effect” an appropriate extra *Set* to *right*, and conversely. However, “side effects” are impossible in FP, so we have to express it explicitly outside of the constituent thermometers *left* and *right*. This is done as follows:

- Let *l2r* be the conversion from the left scale to the right one, and *r2l* the conversion the other way around.
- Out of the request list *in* for the coupled thermometer two request lists $inL, inR$ for the constituent thermometers $left, right$ are produced. An *in*-request $L\,Get$ gives rise to a *Get* in *inL*. An *in*-request $L\,(Set\,m)$ gives a $Set\,m$ followed by a *Get* in *inL*, and a $Set\,(l2r\,y)$ in *inR*, where $y$ is the response to the *Get* in *inL*. Similarly with $R$ and $L$ interchanged.

- The two response lists $\mathit{outL} = \mathit{left}\,\mathit{inL}$ and $\mathit{outR} = \mathit{right}\,\mathit{inR}$ are merged together into one response list $\mathit{out}$ for the couple. The labels in the original request list $\mathit{in}$ are used to decide the order in the final output list, and to decide which responses are due to the extra inserted $\mathit{Get}$ requests and, hence, are to be discarded.

Leaving the retrieval of $\mathit{l2r}$ and $\mathit{r2l}$ to the eventual $\mathit{couple}$, we define here an auxiliary $\mathit{couple}'$ that behaves exactly as the description above:

$$\begin{array}{lcl}
\mathit{couple}'\,(\mathit{left}, \mathit{right})\,\mathit{in} & = & \mathit{out} \\
\text{where} & & \\
\mathit{out} & = & \mathit{merge}\,\mathit{in}\,(\mathit{outL}, \mathit{outR}) \\
(\mathit{outL}, \mathit{outR}) & = & (\mathit{left}\,\mathit{inL},\ \mathit{right}\,\mathit{inR}) \\
(\mathit{inL}, \mathit{inR}) & = & \mathit{split}\,\mathit{in}\,(\mathit{outL}, \mathit{outR}) \quad .
\end{array}$$

Using recursion operator $\mathit{rec}$ and function combinator $\times$, defined by:

$$\begin{array}{lcl}
\mathit{rec}\,f & = & x \text{ where } x = f\,x \\
(f \times g)\ (x, y) & = & (f\,x,\ g\,y) \quad ,
\end{array}$$

the definition of $\mathit{couple}'$ can be given as a one-liner:

$$\mathit{couple}'\,(\mathit{left}, \mathit{right})\,\mathit{in} \quad = \quad \mathit{merge}\,\mathit{in}\,(\mathit{rec}\,((\mathit{left} \times \mathit{right}) \cdot \mathit{split}\,\mathit{in})) \quad .$$

The auxiliary functions $\mathit{split}$ and $\mathit{merge}$ are defined thus:

$$\begin{array}{lcl}
\mathit{split}\,(L\,(\mathit{Set}\,m) : \mathit{xs})\,(\mathit{ys}, \mathit{zs}) & = & \\
\quad (([\mathit{Set}\,m,\, \mathit{Get}]{+\!\!+}) \ \times\ ([\mathit{Set}\,(\mathit{l2r}\,(\mathit{hd}\,\mathit{ys}))]{+\!\!+}))\ \ (\mathit{split}\,\mathit{xs}\,(\mathit{tl}\,\mathit{ys},\ \mathit{zs})) & & \\
\mathit{split}\,(R\,(\mathit{Set}\,m) : \mathit{xs})\,(\mathit{ys}, \mathit{zs}) & = & \\
\quad (([\mathit{Set}\,(\mathit{r2l}\,(\mathit{hd}\,\mathit{zs}))]{+\!\!+}) \ \times\ ([\mathit{Set}\,m,\, \mathit{Get}]{+\!\!+}))\ \ (\mathit{split}\,\mathit{xs}\,(\mathit{ys},\ \mathit{tl}\,\mathit{zs})) & & \\
\mathit{split}\,(L\,\mathit{Get} : \mathit{xs}) & = & (([\mathit{Get}]{+\!\!+}) \ \times\ ([\,]{+\!\!+})) \ \cdot\ \mathit{split}\,\mathit{xs} \\
\mathit{split}\,(R\,\mathit{Get} : \mathit{xs}) & = & (([\,]{+\!\!+}) \ \times\ ([\mathit{Get}]{+\!\!+})) \ \cdot\ \mathit{split}\,\mathit{xs} \\
\mathit{merge}\,(L\,(\mathit{Set}\,x) : \mathit{xs})\,(y : \mathit{ys},\ \mathit{zs}) & = & \mathit{merge}\,\mathit{xs}\,(\mathit{ys}, \mathit{zs}) \\
\mathit{merge}\,(R\,(\mathit{Set}\,x) : \mathit{xs})\,(\mathit{ys},\ z : \mathit{zs}) & = & \mathit{merge}\,\mathit{xs}\,(\mathit{ys}, \mathit{zs}) \\
\mathit{merge}\,(L\,\mathit{Get} : \mathit{xs})\,(y : \mathit{ys},\ \mathit{zs}) & = & y\ :\ \mathit{merge}\,\mathit{xs}\,(\mathit{ys}, \mathit{zs}) \\
\mathit{merge}\,(R\,\mathit{Get} : \mathit{xs})\,(\mathit{ys},\ z : \mathit{zs}) & = & z\ :\ \mathit{merge}\,\mathit{xs}\,(\mathit{ys}, \mathit{zs}) \quad .
\end{array}$$

The reader my check that there is no mutual dependency between any two distinct elements of any two auxiliary or final lists.

There is just one problem left: how to express the conversion functions $\mathit{l2r}$ and $\mathit{r2l}$, used in $\mathit{split}$, in such a way that coupling of thermometers with yet another internal scale is also possible. Clearly, within the definition of $\mathit{couple}'$ it is in general impossible to detect the internal scale of $\mathit{left}$ and $\mathit{right}$. Therefore we have to adapt the model constructed so far: each thermometer also has to provide the conversion functions between its own internal scale (Centigrade or Fahrenheit or whatever) and a fixed scale, say Kelvin. An obvious way to do so, is to extend thermometers into a triple, one component being what we have called thermometer so far, and two components for the conversion functions to and from the fixed scale. Thus we redefine:

$$\begin{array}{lcl} \mathit{centigrade} & = & \mathit{fst}\ \mathit{centigrade}' \\ \mathit{centigrade}' & = & (\mathit{thermometer} \cdot (\mathit{Set}\, 0\ :),\ \ \mathit{c2k},\ \ \mathit{k2c}) \quad . \end{array}$$

Notice that *centigrade* has *not* changed by this redefinition; all programs that already exist and make use of *thermometer* and *centigrade*, keep their semantics. Similarly for *fahrenheit*. Now we can define the required *couple*:

$$\begin{array}{lcl} \mathit{couple}\,(\mathit{left}, \mathit{right}) & = & \mathit{couple}'\,(\mathit{left}', \mathit{right}') \\ \text{where} & & \\ (\mathit{left}',\ \mathit{l2k},\ \mathit{k2l}) & = & \mathit{left} \\ (\mathit{right}',\ \mathit{r2k},\ \mathit{k2r}) & = & \mathit{right} \\ (\mathit{l2r},\ \mathit{r2l}) & = & (\mathit{k2r} \cdot \mathit{l2k},\ \mathit{k2l} \cdot \mathit{r2k}) \quad . \end{array}$$

**Discussion** [*First version — to be rewritten*] It is hard to draw general conclusions from just one test case. Nevertheless we can make some comments. In arbitrary order:

- The FP modeling of objects is, after all, well-known: apart from problems due to timing aspects, there is a translation from CSP processes to list processing functions. It is this translation that we have applied to model thermometers.

- It is easy to give a formal statement of the correctness of the FP model, specifically function *couple*, and it seems easy to prove its correctness formally.

  Within the context of *couple*'s where-part, the specification of *couple'* reads:

$$\begin{array}{l} \mathit{map}\ \mathit{lr2k}' \cdot (\mathit{couple}'\,(\mathit{left}', \mathit{right}') \vartriangle \mathit{id}) \\ = \\ \mathit{thermometer} \cdot (\mathit{Set}\, 273\ \ :) \cdot \mathit{map}\ \mathit{lr2k} \end{array}$$

  where

$$\begin{array}{lcl} (f \vartriangle g)\, x & = & (f\, x,\ \ g\, x) \\ \text{and} & & \\ \mathit{lr2k}\,(L\,(\mathit{Set}\, n)) & = & \mathit{Set}\,(\mathit{l2k}\, n) \\ \mathit{lr2k}\,(R\,(\mathit{Set}\, n)) & = & \mathit{Set}\,(\mathit{r2k}\, n) \\ \mathit{lr2k}\,(L\ \mathit{Get}) & = & \mathit{Get} \\ \mathit{lr2k}\,(R\ \mathit{Get}) & = & \mathit{Get} \\ \text{and} & & \end{array}$$

$$\begin{array}{lcl}
\mathit{lr2k}'\,(m, L\,x) & = & \mathit{l2k}\,m \\
\mathit{lr2k}'\,(m, R\,x) & = & \mathit{r2k}\,m \\
\mathit{lr2k}'\,(m, L\,x) & = & \mathit{l2k}\,m \\
\mathit{lr2k}'\,(m, R\,x) & = & \mathit{r2k}\,m \quad .
\end{array}$$

I wonder what the correctness formulation of the OO model looks like, and how it is proved formally.

- In both the OO approach and the FP approach, all programs that made use of *Centigrade* and *centigrade*, respectively, keep their semantics even when this class / function is changed in the course of defining the coupling of thermometers. This property enhances adaptability and maintainability in both OO and FP.
- The feed-back in the FP model of coupling seems harder to understand and more error-prone than the self-referencing in the OO model of coupling.
- The definitions of the auxiliary functions *split* and *merge* are too long to compare favourably with the OO modeling of feed-back.
- For an experienced FP fanatic the FP model is straightforward. However, it takes some time and effort (and intelligence) to become experienced in FP. I suspect that some people simply never get to that level, whereas they do succeed in reaching an acceptable level in imperative and OO programming.
- Using monads and folds, some definitions in the FP model and in the specification above may be simplified. However, the programmer should be even more experienced in order to think of those simplifications.

## References

[1] M.M. Fokkinga. Programming language concepts — the lambda calculus approach. In P.R.J. Asveld and A. Nijholt, editors, *Essays on Concepts, Formalism, and Tools*, volume 42 of *CWI Tract*, pages 129–162. CWI, Amsterdam, 1987.

[2] P. Henderson. *Object-Oriented Specification and Design with C++*. McGraw-Hill International (UK), 1993.

```
|| A n   F P   m o d e l  o f   C O U P L E D   T H E R M O M E T E R S
|| Maarten Fokkinga, April 1995

|| For an explanation, see my paper "OO versus FP -- a little case study".
|| The type specifications and some auxiliary functions are at the end.

|| ============================ Tests ===================================

in =
  [Set 747, Set 0, Get, Get, Set 37, Get, Get, Set 100, Set 100, Get, Get]
  ++ repeat Get
inLR =
  [L Get, R (Set 100), R Get, L Get, L (Set 100), R Get, L Get]
  ++ repeat (L Get)

tst   = thermometer in
        || = ([0, 0, 37, 37, 100, 100] ++ repeat 100)
tstC  = centigrade (drop 2 in)
        || = ([0, 0, 37, 37, 100, 100] ++ repeat 100)
tstCF = couple (centigrade',fahrenheit') inLR
        || = ([0, 100, 37, 32, 100] ++ repeat 100)
tstCC = couple (centigrade',centigrade') inLR
        || = ([0, 100, 100, 0, 100] ++ repeat 100)

|| ======================== Main definitions ============================

|| ---------------- g e n e r i c  t h e r m o m e t e r ---------------

request * ::= Set * | Get

thermometer =
  f undef
  where
  f m (Set n: xs) = f n xs
  f m (Get: xs)   = m: f m xs

|| ---------------------- s p e c i a l i s a t i o n --------------------

|| || Initial design:
|| centigrade = thermometer . (Set 0:)
|| fahrenheit = thermometer . (Set 32:)

centigrade = fst3 centigrade'
centigrade' = (thermometer . (Set 0:), c2k, k2c)
k2c k = k - 273
```

```
c2k c = c + 273

fahrenheit  = fst3 fahrenheit'
fahrenheit' = (thermometer . (Set 32:), f2k, k2f)
k2f k = (k - 273) * 9 div 5 + 32
f2k f = (f - 32) * 5 div 9 + 273

|| ----------------------- c o u p l i n g ----------------------------

requestLR * ::= L (request *) | R (request *)

|||| Initial design (correct if  l2r  and  r2l  are globally known):
||
||couple' (left, right) in
|| = out
||   where
||   out = murge in (outL, outR)
||   (outL, outR) = (left inL, right inR)
||   (inL, inR)   = split in (outL, outR)
|| = murge in (rec ((left $xXx right) . split in))
||
||split (L (Set m): xs) (ys,zs) =
||  (([Set m, Get]++)  $xXx  ([Set (l2r (hd ys))]++))  (split xs (tl ys, zs))
||split (R (Set m): xs) (ys,zs) =
||  (([Set (r2l (hd zs))]++)  $xXx  ([Set m, Get]++))  (split xs (ys, tl zs))
||split (L Get: xs) =  (([Get]++)  $xXx  ([]++))  .  split xs
||split (R Get: xs) =  (([]++)  $xXx  ([Get]++))  .  split xs

couple (left, right) =
  couple'' (l2r,r2l) (left', right')
  where
  (left',  l2k, k2l) = left
  (right', r2k, k2r) = right
  (l2r, r2l)         = (k2r.l2k,  k2l.r2k)
couple'' (l2r,r2l) (left, right) in =
  murge in (rec ((left $xXx right) . split'' (l2r,r2l) in))

split'' (f,g) (L (Set m): xs) (ys,zs) =
  (([Set m, Get]++) $xXx ([Set (f (hd ys))]++)) (split'' (f,g) xs (tl ys, zs))
split'' (f,g) (R (Set m): xs) (ys,zs) =
  (([Set (g (hd zs))]++) $xXx ([Set m, Get]++)) (split'' (f,g) xs (ys, tl zs))
split'' fg (L Get: xs) =  (([Get]++)  $xXx  ([]++))  .  split'' fg xs
split'' fg (R Get: xs) =  (([]++)  $xXx  ([Get]++))  .  split'' fg xs

murge (L (Set x): xs) (y:ys, zs) = murge xs (ys,zs)
murge (R (Set x): xs) (ys, z:zs) = murge xs (ys,zs)
murge (L Get:     xs) (y:ys, zs) = y: murge xs (ys,zs)
murge (R Get:     xs) (ys, z:zs) = z: murge xs (ys,zs)

|| ============================ Aux and types ============================

rec f            = x where x = f x
(f $xXx g) (x,y) = (f x, g y)
fst3 (a,b,c)     = a

genericThermometer == [request num] -> [num]
scaledThermometer  == (genericThermometer, num->num, num->num)
coupledThermometer == [requestLR num] -> [num]

thermometer, centigrade, fahrenheit :: genericThermometer
centigrade', fahrenheit'            :: scaledThermometer

couple ::  (scaledThermometer, scaledThermometer) -> coupledThermometer
couple'':: (num->num, num->num) ->
           (genericThermometer, genericThermometer) ->
           coupledThermometer

murge :: [requestLR num] -> ([num],[num]) -> [num]

|| ================== end of Coupled Thermometers ======================
```