An extension of the program derivation format

A.J.M. van Gasteren and A. Bijlsma
lex.bijlsma@acm.org

June 30, 1998

Abstract

A convention is proposed for embedding program statements into Dijkstra's calculus, with the aim of simplifying the stepwise construction of programs.

Keywords: program derivation, predicate calculus, data refinement.

1 INTRODUCTION

In this paper we propose an extension to the proof format commonly used for calculational program derivation. This format, which was invented by W.H.J. Feijen, is becoming widely used in the computing community and has proved itself very useful for program derivation as well as the construction of mathematical proofs in general. In program derivation, it is usually employed together with the axiom of assignment to derive program fragments consisting of a single concurrent assignment in a fixed context. The extension is intended to facilitate the derivation of programs consisting of a sequence of assignments or procedure calls. To this end, we show how program statements may be successively embedded in an ongoing derivation, the intended result of which is the stepwise construction of the program in the course of the calculation.

2 REVIEW OF THE PROOF FORMAT

In the usual version of the proof format we find derivations like

$$\begin{array}{ll} & P \\ \equiv & \{\text{explanation of } [P \equiv Q]\,\} \\ & Q \\ \Leftarrow & \{\text{explanation of } [Q \Leftarrow R]\,\} \\ & R \ , \end{array}$$

where P, Q, R denote predicates on a fixed space, and the brackets $[\ldots]$ denote universal quantification over that space, as in Dijkstra *et al.* (1990). From this we may draw the conclusion $[P \Leftarrow R]$. More generally, for any structures P, Q, R and relational binary operators $\oplus$ and $\otimes$ the derivation

$$\begin{array}{ll} & P \\ \oplus & \{\text{explanation of } [P \oplus Q]\,\} \\ & Q \\ \otimes & \{\text{explanation of } [Q \otimes R]\,\} \\ & R \end{array}$$

allows us to conclude $[P\ (\oplus)\circ(\otimes)\ R]$, where $(\oplus)\circ(\otimes)$ is the relational composition of binary relations $\oplus$ and $\otimes$. This conclusion is only useful if composition $(\oplus)\circ(\otimes)$ can be simplified, i.e. is equal to or implies a single familiar operator. Composition $(\Rightarrow)\circ(\Leftarrow)$, for instance, is decidedly not useful. The most commonly used compositions are composition with equality and continued compositions of a transitive operator:

$$\begin{array}{lcll}
(=)\circ(\oplus) & = & (\oplus) & , \\
(\oplus)\circ(=) & = & (\oplus) & , \\
(\oplus)\circ(\oplus) & \Rightarrow & (\oplus) & \text{for transitive } \oplus \quad , \\
(\oplus)\circ(\oplus) & = & (\oplus) & \text{for transitive and reflexive } \oplus \quad .
\end{array}$$

Here is a simplification that is perhaps less familiar: if we write $x =_m y$ for $x \ \mathsf{mod}\ m = y \ \mathsf{mod}\ m$, the derivation

$$\begin{array}{ll}
 & x \\
=_m & \{\text{explanation of } [x \ \mathsf{mod}\ m = y \ \mathsf{mod}\ m]\,\} \\
 & y \\
=_n & \{\text{explanation of } [y \ \mathsf{mod}\ n = z \ \mathsf{mod}\ n]\,\} \\
 & z
\end{array}$$

admits the conclusion $[x \ \mathsf{mod}\ (m \ \mathsf{gcd}\ n) = z \ \mathsf{mod}\ (m \ \mathsf{gcd}\ n)]$. Another slightly unusual simplification is

$$(\not\equiv)\circ(\not\equiv) \ = \ (\equiv) \quad .$$

An example of its usefulness is the proof of Theorem 4 of Bijlsma *et al.* (1996).

3 DERIVATIONS CONTAINING PROGRAM STATEMENTS

For structure transformer f and relational binary operator $\oplus$, we propose to admit into program derivations steps like

$$\begin{array}{ll}
 & P \\
\oplus & \lhd f \rhd \\
 & \quad \{\text{explanation of } [f.P \oplus Q]\,\} \\
 & Q
\end{array}$$

which, as the hint suggests, will signify the validity of $[f.P \oplus Q]$. We propose to pronounce the above step, in the case where $\oplus$ is $\Leftarrow$, as "P follows by f from Q", and similarly in other cases. As to the combination of such steps,

$$\begin{array}{ll}
 & P \\
\oplus & \lhd f \rhd \\
 & \quad \{\text{explanation of } [f.P \oplus Q]\,\} \\
 & Q \\
\otimes & \lhd g \rhd \\
 & \quad \{\text{explanation of } [g.Q \otimes R]\,\} \\
 & R
\end{array}$$

admits only the conclusion $[f.P {\oplus} Q] \wedge [g.Q {\otimes} R]$, which cannot be simplified any further without knowledge of the properties of the operators and structure transformers involved. However, if the operators are $\equiv$, it follows that $[g.(f.P) \equiv R]$; if the operators are $\Leftarrow$ and g is monotonic, it follows that $[g.(f.P) \Leftarrow R]$.

In this paper, we limit ourselves to a special case where the insertion of structure transformers yields particularly useful conclusions. From now on f will be a particular predicate transformer, namely the weakest precondition of a program statement S . For this particular purpose, we prefer to write $\triangleleft\, S\, \triangleright$ rather than $\triangleleft\, wp.S\, \triangleright$. As promised in the Introduction, this allows us to write derivations like

$$\begin{array}{ll} & P \\ \Leftarrow & \triangleleft\, S\, \triangleright \\ & \quad \{\text{explanation of } [wp.S.P \Leftarrow Q]\,\} \\ & Q \\ \equiv & \triangleleft\, T\, \triangleright \\ & \quad \{\text{explanation of } [wp.T.Q \equiv R]\,\} \\ & R \quad , \end{array}$$

and draw the conclusion

$$\{R\} \;\; T;\; S \;\; \{P\} \quad .$$

Observe that the statements of the program are discovered consecutively in the course of a single derivation. Because

$$\begin{array}{ll} & P \\ \Leftarrow & \quad \{\text{explanation of } [P \Leftarrow Q]\,\} \\ & Q \end{array}$$

is equivalent to

$$\begin{array}{ll} & P \\ \Leftarrow & \triangleleft\, \mathit{skip}\, \triangleright \\ & \quad \{\text{explanation of } [P \Leftarrow Q]\,\} \\ & Q \quad , \end{array}$$

steps introducing a program statement may be freely intermixed with classical steps expressing a strengthening or an equivalence.

Example 1 The task is to write a program that cyclically rotates the values of integer variables x, y, z and uses neither auxiliary variables nor multiple assignments. The analogous program with two variables occurs in many textbooks of programming (e.g. Dijkstra *et al.* (1984)) as an example of how to use the semantics of assignment, but it is always given as a task for a posteriori verification, never derived. The specification of the problem is

$$\begin{array}{l} \{\, \mathit{true}\, \} \\ \mathit{rot}.x.y.z \\ \{\, x = y_\bullet \;\wedge\; y = z_\bullet \;\wedge\; z = x_\bullet\, \} \quad , \end{array}$$

where we have followed the convention of denoting the original value of x by $x_\bullet$ (pronounce ‘x old’). (This can be seen as an application of the traditional ‘specification variable’ or ‘logical constant’ approach where the precondition is implicitly extended with terms like $x = x_\bullet$.)

Without introducing statements into derivations there is really no good way to tackle this. With statements, however, we may derive

$x = y_\bullet \ \wedge \ y = z_\bullet \ \wedge \ z = x_\bullet$

$\equiv \quad \lhd\, x := x + y + z \,\rhd$

  {to bring y and $y_\bullet$ together in a single conjunct}

$x + y + z = y_\bullet \ \wedge \ y = z_\bullet \ \wedge \ z = x_\bullet$

$\equiv \quad \lhd\, y := -x + y - z \,\rhd$

  {to create a term $y = y_\bullet$ }

$y = y_\bullet \ \wedge \ -x + y - z = z_\bullet \ \wedge \ z = x_\bullet$

$\equiv \quad \lhd\, z := -x + y - z \,\rhd$

  {to create a term $z = z_\bullet$ }

$y = y_\bullet \ \wedge \ z = z_\bullet \ \wedge \ -x + y - z = x_\bullet$

$\equiv \quad \lhd\, x := -x + y - z \,\rhd$

  {to create a term $x = x_\bullet$ }

$y = y_\bullet \ \wedge \ z = z_\bullet \ \wedge \ x = x_\bullet$.

(Note that the first step in this derivation is rather arbitrary: it should satisfy the criteria that it introduces y into the first conjunct and does not remove x from it, but apart from these we have a lot of choice. After the first step, however, there is no more choice and the calculation is guided entirely by the wish to create the conjuncts of the precondition one by one.)

We conclude that the specification is satisfied by the program

```
  { $true$ }
  $x := -x + y - z$
; $z := -x + y - z$
; $y := -x + y - z$
; $x := x + y + z$
  { $x = y_\bullet \ \wedge \ y = x_\bullet \ \wedge \ z = x_\bullet$ }  .
```

□

4 DERIVATIONS CONTAINING COERCION STATEMENTS

For every predicate B, a program statement S is defined by the condition that, for every predicate R,

$$[wp.S.R \ \equiv \ (R \ \Leftarrow \ B)].$$

Such a statement S is called a *coercion statement* in Morgan (1990). We shall denote the coercion statement corresponding to a boolean expression B by B as well, as no confusion threatens. The coercion statement behaves like **magic** in case $\neg B$ holds, and hence it is not implementable; its value lies in the fact that Hoare triples like

$$\{P\} \ \ B; \ S \ \ \{Q\}$$

and

$$\{P\} \ \ \neg B; \ T \ \ \{Q\}$$

can be combined into

```
{ $P$ }
if $B \rightarrow$   $S$
[] $\neg B \rightarrow$ $T$
fi
{ $Q$ } .
```

We shall see several cases of this use of coercion statements to produce selection statements in the next sections. In the present section, however, we concentrate on derivations that contain *only* coercion statements, i.e. no statements of other kinds. It will be seen that these can be used as a way to introduce local hypotheses into proofs that may have nothing to do with programming at all.

An important ingredient of natural deduction systems is their ability to introduce local hypotheses into a proof. Equational reasoning, as commonly understood, lacks a facility for doing this – possibly the most important cause of the continued popularity of otherwise awkward and unwieldy proof systems based on natural deduction.

Consider a proof step where a coercion statement B is introduced, say

```
   $P$
$\Leftarrow$  $\triangleleft B \triangleright$
      {explanation of $[(P \Leftarrow B) \Leftarrow Q]$ }
   $Q$ .
```

Here the hint can be simplified to $[P \Leftarrow B \wedge Q]$. As to the composition of such steps, observe that

```
   $P$
$\Leftarrow$  $\triangleleft B \triangleright$
      {explanation of $[P \Leftarrow B \wedge Q]$ }
   $Q$
$\Leftarrow$  $\triangleleft C \triangleright$
      {explanation of $[Q \Leftarrow C \wedge R]$ }
   $R$
```

admits conclusion

$$[P \Leftarrow B \wedge C \wedge R] \ .$$

Using this technique to prove an implication gives us the opportunity to choose among several different calculations that differ in the way the antecedent is distributed over the hints. Conversely, to draw a conclusion from such a derivation, one has to gather the conjuncts of the antecedent from the proof steps in which a hypothesis was introduced. This is made easy by the use of the conspicuous symbols $\triangleleft \ldots \triangleright$.

Remark This method of introducing hypotheses into a transformational proof has aims similar to the generalized window inference proposed by Grundy (1996); however, our method requires much less of a departure from normal practice. Some other authors, e.g. Gries *et al.* (1993, sec. 4.2), use a notation like

```
   $P$
$\equiv$     {**Assumption** $X$ }
   $Q$ ,
```

which requires the validity of

$$[(P \equiv Q) \;\Leftarrow\; X] \quad .$$

This is not the same as a step of the form

$$\begin{array}{ll} & P \\ \equiv & \quad \lhd X \rhd \\ & \qquad \{\text{explanation}\} \\ & Q \end{array}$$

which requires the stronger

$$[(P \Leftarrow X) \;\equiv\; Q] \quad .$$

This stronger property has the advantage that Q is precisely the weakest precondition of the program obtained: there is no 'slack' in the calculation. If, however, the operator is a consequence ($\Leftarrow$) rather than an equivalence ($\equiv$), the proof obligations in both styles coincide.
□

When using coercion statements in combination with other statements, one must be careful to remember that sequential composition is not commutative. For instance, a calculation like

$$\begin{array}{ll} & \mathit{false} \\ \equiv & \quad \lhd b \rhd \\ & \qquad \{\, (b \Rightarrow \mathit{false}) \;\equiv\; \neg b \,\} \\ & \neg b \\ \equiv & \quad \lhd b := \neg b \rhd \\ & \qquad \{\text{axiom of assignment}\} \\ & b \end{array}$$

does *not* imply the validity of

$$\{b\} \;\; b := \neg b \;\; \{\mathit{false}\} \quad .$$

One situation in which the successive introduction of coercions is particularly useful is in the invention of the right way to strengthen the invariant of a repetition. This is illustrated in the following example[1]

Example 2 Given is an integer array $a[0\,..\,N)$, where $N \geq 0$. We are to determine the number of unordered pairs of distinct indices whose array values have a nonnegative product. Formally, we are to establish the postcondition

$$r = \langle \# p, q : 0 \leq p < q < N : a.p * a.q \geq 0 \rangle \quad .$$

For our first approximation, we replace a constant by a variable and obtain invariants

$$\begin{array}{lll} P0 & : & 0 \leq n \leq N \quad , \\ P1 & : & r = \langle \# p, q : 0 \leq p < q < n : a.p * a.q \geq 0 \rangle \quad , \end{array}$$

which is initialized by $n, r := 0, 0$. In order to decide how we can maintain $P1$ under $n := n{+}1$, we calculate as follows:

[1]The example given is, in fact, exercise 4.3.3 of Kaldewaij (1990).

$$
\begin{array}{ll}
 & P1 \\
\equiv & \lhd\ n := n+1\ \rhd \\
 & \quad \{\text{for progress}\} \\
 & r = \langle \#p,q : 0 \le p < q < n+1 : a.p * a.q \ge 0 \rangle \\
\equiv & \lhd\ r := r'\ \rhd \\
 & \quad \{\text{to reestablish } P1\} \\
 & r' = \langle \#p,q : 0 \le p < q < n+1 : a.p * a.q \ge 0 \rangle \\
\Leftarrow & \lhd\ P0 \ \wedge\ n \ne N\ \rhd \\
 & \quad \{\text{split off } q = n\} \\
 & r' = \langle \#p,q : 0 \le p < q < n : a.p * a.q \ge 0 \rangle + \langle \#p : 0 \le p < n : a.p * a.n \ge 0 \rangle \\
\Leftarrow & \lhd\ P1\ \rhd \\
 & \quad \{\text{we now start leaving out the domain } 0 \le p < n\} \\
 & r' = r + \langle \#p :: a.p * a.n \ge 0 \rangle \\
\equiv & \quad \{\text{case analysis}\} \\
 & (a.n > 0 \ \wedge\ r' = r + \langle \#p :: a.p \ge 0 \rangle) \\
 & \vee\ (a.n = 0 \ \wedge\ r' = r + n) \\
 & \vee\ (a.n < 0 \ \wedge\ r' = r + \langle \#p :: a.p \le 0 \rangle) \\
\Leftarrow & \lhd\ s = \langle \#p :: a.p \ge 0 \rangle \ \wedge\ t = \langle \#p :: a.p \le 0 \rangle\ \rhd \\
 & \quad \{\} \\
 & (a.n > 0 \ \wedge\ r' = r + s) \ \vee\ (a.n = 0 \ \wedge\ r' = r + n) \ \vee\ (a.n < 0 \ \wedge\ r' = r + t) \quad .
\end{array}
$$

The predicate following the final $\Leftarrow$ is a good candidate for a strengthening of the invariant; notice that this predicate forms a context for the above calculation as a whole, but is produced by the calculation itself, making it impractical to insist upon stating such conditions in advance.
□

5 DERIVATIONS CONTAINING METHOD CALLS

In object-oriented programs, one uses abstract classes or interfaces that are essentially abstract data types whose implementation is not statically known. The only way to derive programs using such interfaces is by agreeing on a specification that future implementers must respect. There are several ways to specify a class; here we follow Morgan (1990) in associating with every object one or more *thought variables*, taken from some mathematically tractable domain, and specifying the methods in the interface by pre- and postconditions expressed in the thought variables. The relation of these abstract specifications to the implementing code is that the latter must be a *data refinement* of the former.

Example 3 Consider the abstract data type queue, given by the following interface. An object q of type queue is described by a thought variable $q.s$ denoting a finite list of values of some component type. For a queue q, boolean function $q.empty$ is specified in terms of $q.s$ by

$$q.empty \ \equiv\ q.s = [\,] \quad .$$

Function $q.head$, returning a value of the component type, is specified by

$$q.s \ne [\,] \ \Rightarrow\ q.head = hd.(q.s) \quad ,$$

where hd denotes the standard list-processing function that, when applied to a nonempty list, returns the value of its first term. Procedure call $q.add(a)$ can be specified by a Hoare triple

$$\{true\} \ \ q.add(a) \ \ \{q.s = q.s_\bullet \mathbin{+\!\!+} [a]\} \quad ,$$

but it is more convenient, and semantically equivalent, to think of it as an assignment to the abstract variable

$$q.s := q.s +\!\!+ [a] \quad .$$

It is often the case that procedure calls can be viewed as abstract assignments; for nondeterministic specifications, however, this point of view presents difficulties and we need the power of a general procedure call proof rule (e.g. Bijlsma *et al.* (1989)). Finally, procedure call $q.remove$ can be specified by a Hoare triple

$$\{q.s \neq [\,]\} \;\; q.remove \;\; \{q.s = tl.(q.s_\bullet)\}$$

or by the abstract assignment

$$q.s := tl.(q.s)$$

where tl is the standard list-processing function that applies to a nonempty list and returns that list minus its first term. Because tl is a partial function, the weakest precondition of the abstract assignment contains a conjunct $q.s \neq [\,]$.

The problem we now address is how to merge two given ascending queues into a single one. Formally, this may be specified by

$$\begin{array}{l} \{\, asc.(p.s) \;\wedge\; asc.(q.s) \,\} \\ merge.p.q.r \\ \{\, R: \quad r.s = sort.(p.s_\bullet +\!\!+ q.s_\bullet) \,\} \quad . \end{array}$$

We achieve our purpose by a repetition with a tail invariant, viz.

$$P0: \quad sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ sort.(p.s +\!\!+ q.s) \quad .$$

The choice of $P0$ is guided by the considerations that, first, $P0$ is easy to establish since

$$p.s = p.s_\bullet \;\wedge\; q.s = q.s_\bullet \;\Rightarrow\; (r.s := [\,]).P0 \quad ,$$

and, secondly, $P0$ helps to attain the required postcondition since

$$P0 \;\wedge\; p.s +\!\!+ q.s = [\,] \;\Rightarrow\; R \quad .$$

As a further constraint on the design, we decide to add the precondition as an invariant, viz.

$$P1: \quad asc.(p.s) \;\wedge\; asc.(q.s) \quad .$$

The rationale behind this is that, the given queues being already sorted, it would be a waste not to exploit this fact. Progress will consist in extending r until it is as long as $p.s_\bullet +\!\!+ q.s_\bullet$ under invariance of $P0\,..\,1$. We calculate as follows:

$$\begin{array}{ll} & P0 \\ \equiv & \quad \{\text{definition of } P0\} \\ & sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ sort.(p.s +\!\!+ q.s) \\ \equiv & \triangleleft\, r.add(a) \,\triangleright \\ & \quad \{\text{for progress}\} \\ & sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ [a] +\!\!+ sort.(p.s +\!\!+ q.s) \\ \Leftarrow & \quad \{\text{property of } sort\text{, see (1) below}\} \\ & sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ sort.([a] +\!\!+ p.s +\!\!+ q.s) \;\wedge\; a = min.([a] +\!\!+ p.s +\!\!+ q.s) \end{array}$$

$\equiv \quad \lhd\, p.remove \,\rhd$
  $\{$preparing to get rid of $[a]\,\}$
  $sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ sort.([a] +\!\!+ tl.(p.s) +\!\!+ q.s)$
  $\wedge\ a = min.([a] +\!\!+ tl.(p.s) +\!\!+ q.s)\ \wedge\ p.s \neq [\,]$
$\Leftarrow$ $\{$to combine $[a]$ with $tl.(p.s)$, heading for $P0\,\}$
  $sort.(p.s_\bullet +\!\!+ q.s_\bullet) = r.s +\!\!+ sort.(p.s +\!\!+ q.s)$
  $\wedge\ a = min.(p.s +\!\!+ q.s)\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a$
$\equiv$ $\{$definition of $P0\,\}$
  $P0\ \wedge\ a = min.(p.s +\!\!+ q.s)\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a$
$\equiv$ $\{$property of min, see (3) below$\}$
  $P0\ \wedge\ a = min.(p.s)\ \downarrow\ min.(q.s)\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a$
$\Leftarrow$ $\{$property of min, see (2) below$\}$
  $P0..1\ \wedge\ a = a\ \downarrow\ b\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a\ \wedge\ q.s \neq [\,]\ \wedge\ hd.(q.s) = b$
$\equiv$ $\{$property of $\downarrow\,\}$
  $P0..1\ \wedge\ a \leq b\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a\ \wedge\ q.s \neq [\,]\ \wedge\ hd.(q.s) = b$
$\Leftarrow \quad \lhd\, a \leq b \,\rhd$
  $\{$can be checked in constant time$\}$
  $P0..1\ \wedge\ p.s \neq [\,]\ \wedge\ hd.(p.s) = a\ \wedge\ q.s \neq [\,]\ \wedge\ hd.(q.s) = b$
$\Leftarrow \quad \lhd\, a, b := p.head,\ q.head \,\rhd$
  $\{\}$
  $P0..1\ \wedge\ p.s \neq [\,]\ \wedge\ q.s \neq [\,]$.

Thus we have proved

```
  { $P0..1 \wedge p.s \neq [\,] \wedge q.s \neq [\,]$ }
  $a, b := p.head,\ q.head$
; $a \leq b$
; $p.remove$
; $r.add(a)$
  { $P0..1$ } .
```

By symmetry, this gives

```
  { $P0..1 \wedge p.s \neq [\,] \wedge q.s \neq [\,]$ }
  $a, b := p.head,\ q.head$
; if $a \leq b \rightarrow$ $p.remove$; $r.add(a)$
  [] $b \leq a \rightarrow$ $q.remove$; $r.add(b)$
  fi
  { $P0..1$ } .
```

This is the heart of the merge algorithm. The only ingredient still missing, the case where one of the queues has become empty, is treated the same way. In the derivation, some elementary properties of list calculus were used. These are

$$sort.([\alpha]+\!\!+\sigma) = [\alpha]+\!\!+sort.\sigma \quad \equiv \quad \alpha = \ min.([\alpha]+\!\!+\sigma) \ , \tag{1}$$

$$\sigma \neq [\,]\ \wedge\ asc.(\sigma) \quad \Rightarrow \quad hd.\sigma = min.\sigma \ , \tag{2}$$

$$min.(\sigma+\!\!+\tau) \quad = \quad min.\sigma\ \downarrow\ min.\tau \ . \tag{3}$$

□

6 APPLICATION: SIMULATING FILE OPERATIONS

6.1 Specification of the ISO Pascal file type

In this section, we apply the techniques developed earlier in the context of a realistic example. Indeed, we have chosen an example that is well-known to anyone who has ever struggled to convert programs involving ISO Pascal file operations to some nonstandard Pascal version – an experience by now familiar to a sizable proportion of the earth's population. In order to derive a solution to this problem, we regard ISO Pascal files as an abstract data type, disregarding the fact that Pascal compilers will not support this. We specify this abstract data type by giving a *model*, i.e. we define a number of thought variables of mathematically well-understood types in terms of which the operations on ISO Pascal files can be described. Let a component type T be given. Our model consists of four thought variables, namely

$P : T^*$	(the file prefix, consisting of the items already read)
$S : T^*$	(the file suffix, consisting of the items not yet read)
$Q : \{in, out\}$	(whether the file is opened for input or for output)
$B : T \cup \{\perp\}$	(the file buffer)

The values of these four variables are not quite independent: they are linked by the *type invariant*

$$Q = out \;\Rightarrow\; S = [\,] \quad . \tag{4}$$

The value $\perp$ (pronounced 'bottom') is added because the ISO Pascal standard sometimes requires the buffer to become undefined.

The abstract data type for ISO Pascal files is defined by listing its operations and specifying these by pre- and postconditions in terms of the four variables P, S, Q, B. Here we shall concentrate on specifying and implementing a single operation, $Reset$. Readers wishing to see the specification and implementation of the full ISO Pascal type may consult Bijlsma (1997). The specification of $Reset$ reads as follows:

pre: $true$
post: $P = [\,] \;\wedge\; S = P_{\bullet} \mathbin{+\!\!+} S_{\bullet} \;\wedge\; Q = in \;\wedge\; B = Hd.S$

Function Hd is similar to, but slightly more general than, the function hd usually provided in functional programming languages. It is defined inductively by

$$\begin{array}{rcl} Hd.[\,] & = & \perp \\ Hd.([a] \mathbin{+\!\!+} as) & = & a \end{array}$$

The proof obligation that this specification is compatible with type invariant (4) is left to the reader.

6.2 Specification of a simple file type

In this section, we specify a simpler file type that can be regarded as a greatest common divisor of several commercial implementations of Pascal. Here the model consists of only two variables, namely

$p : T^*$	(the file prefix, consisting of the items already read)
$s : T^*$	(the file suffix, consisting of the items not yet read)

In the same notation as the previous section, the relevant operations can be specified as follows:

reset pre: *true*
post: $p = [\,] \;\wedge\; s = p_\bullet +\!\!+\, s_\bullet$

read(x) pre: $s \neq [\,]$
post: $p = p_\bullet +\!\!+\, [Hd.s_\bullet] \;\wedge\; s = Tl.s_\bullet \;\wedge\; x = Hd.s_\bullet$

eof pre: *true*
ret: $s = [\,]$

However, we are not yet quite satisfied with this form. The reason is that specifying a relation by means of a precondition/postcondition pair is the easiest way if we are asked to *implement* that relation; however, if we are to *use* the procedure described by a relation as a given building block to be used in programming some higher-level module, it is far more convenient to have the specification in a predicate-transformer form. This is why, traditionally, programming exercises are specified by a pair of predicates, while programming constructs are specified by a predicate transformer. Now in a state space with coordinate vectors x, y, any specification of the form

pre: $def.E$
post: $x = x_\bullet \;\wedge\; y = E_\bullet$

has the same weakest precondition as, hence can be identified with, the assignment $y := E$.

Therefore we may reformulate our specification as follows:

$$reset \;=\; p, s := [\,], p+\!\!+s \tag{5}$$

$$read(x) \;=\; \{s \neq [\,]\};\; p, s, x := p+\!\!+[Hd.s],\; Tl.s,\; Hd.s \tag{6}$$

$$eof \;\equiv\; s = [\,] \tag{7}$$

The statement $\{s \neq [\,]\}$ is a so-called assertion statement, whose semantics is defined by

$$wp.\{Q\}.R \;\equiv\; Q \;\wedge\; R \quad .$$

In other words, assertion statement $\{Q\}$ behaves like *skip* when boolean condition Q is satisfied, and aborts otherwise. See Morgan (1990), Morris (1989) for further discussion. This is not to be confused with the coercion statement introduced in an earlier section, whose semantics is defined by

$$wp.Q.R \;\equiv\; (Q \Rightarrow R) \quad .$$

6.3 Derivation of the implementation code

It is our goal to implement the specification of the ISO file operations in terms of the simpler operations of the previous section. Now the theory of data refinement (Morgan 1990) (Morgan *et al.* 1990) (Morris 1989) states that data refining an abstract specification

pre: U, post: V

is achieved by choosing an *abstraction invariant* A linking the abstract variables a and the concrete variables c, and then constructing code S that does not mention a and satisfies

$$\{\langle \exists a :: U \wedge A \rangle\} \;\; S \;\; \{\langle \exists a :: V \wedge A \rangle\} \quad . \tag{8}$$

As a practical observation, we prefer to work without the quantifiers and just construct some code S satisfying

$$\{U \wedge A\} \;\; S \;\; \{V \wedge A\} \quad ,$$

where S may mention a but only as a *ghost variable*, that is to say that in S, the variables from a may not occur in statements that change the value of any other variables. If we have constructed an S satisfying these restrictions, it can be turned into a program fragment satisfying (8) by simply leaving out all statements that mention a. For the problem at hand, we take as concrete variables the sequences p and s from the preceding section, augmented by the following variables:

$$\begin{array}{ll} d, e: & \mathit{Boolean} \\ f, g: & T \end{array}$$

Now we propose the following abstraction invariant:

$$\begin{array}{ll} A0: & d \;\Rightarrow\; P = p \;\wedge\; S = s = [\,] \\ A1: & \neg d \;\Rightarrow\; p = P +\!\!+ [f] \;\wedge\; S = [f] +\!\!+ s \\ A2: & e \;\Rightarrow\; B = g \\ A3: & \neg e \;\Rightarrow\; B = \bot \end{array}$$

We shall use A as an abbreviation for $A0 \wedge A1 \wedge A2 \wedge A3$. The heuristics behind the choice of A are as follows: in the first place, in our simple file type there is no equivalent of the file buffer B. In order to simulate B, we introduce variable g. We assume that value $\bot$ is not actually implemented, so we simulate that by letting g be of type T rather than $T \cup \{\bot\}$, and letting $\neg e$ signal[2] when B should have the value $\bot$. Now a straightforward simulation would simulate P and S by p and s respectively, but the difficulty with this is that the semantics of *Reset* requires setting the file buffer to the first unread element. Our simple file system affords no knowledge of unread elements, so the only recourse is reading one extra element and preserving its value in f. This strategy implies *eof* may become true one step sooner than its ISO file analogue *Eof*, so we add another boolean d to simulate *Eof*.

Observe that A nowhere mentions Q, so the abstract state is not determined by the concrete state. Therefore this abstraction invariant cannot be replaced by the better-known technique of abstraction *functions*.

Now we are ready to derive the implementation of the ISO file operation *Reset*. Let R denote the postcondition given for *Reset*, i.e.

$$R: \quad P = [\,] \;\wedge\; S = P_{\bullet} +\!\!+ S_{\bullet} \;\wedge\; Q = \mathit{in} \;\wedge\; B = \mathit{Hd}.S \quad .$$

Code is derived separately for postconditions $R \wedge S{=}[\,]$ and $R \wedge S{\neq}[\,]$, as we are more or less forced to do on account of the shape of $A0$ and $A1$. As both derivations are similar, we give only the derivation for the case $R \wedge S \neq [\,]$ and refer the reader to Bijlsma (1997) for the other case.

[2]Actually, it turns out that e is superfluous, as we shall see at the end of this section. It is not necessary to observe this at the present stage of development.

$R \wedge S \neq [\,] \wedge A$

$\equiv$ {definitions of R, A, and Hd}

$P = [\,] \wedge p = [f] \wedge S = [f] +\!\!+ s = P_\bullet +\!\!+ S_\bullet \wedge \neg d \wedge Q = in \wedge e \wedge B = f = g$

$\equiv$ $\triangleleft\, P, S, Q, B := [\,], [f] +\!\!+ s, in, f \,\triangleright$
{eliminate abstract variables by explicit assignment}

$p = [f] \wedge [f] +\!\!+ s = P_\bullet +\!\!+ S_\bullet \wedge \neg d \wedge e \wedge f = g$

$\equiv$ $\triangleleft\, d, e, g := false, true, f \,\triangleright$
{eliminate booleans and pseudo-buffer g by explicit assignment}

$p = [f] \wedge [f] +\!\!+ s = P_\bullet +\!\!+ S_\bullet$

$\equiv$ $\triangleleft\, read(f) \,\triangleright$
{applying (6) eliminates f}

$s \neq [\,] \wedge p = [\,] \wedge s = P_\bullet +\!\!+ S_\bullet$

$\equiv$ $\triangleleft\, \neg eof \,\triangleright$
{applying (7) moves s to antecedent}

$s \neq [\,] \Rightarrow p = [\,] \wedge s = P_\bullet +\!\!+ S_\bullet$

$\Leftarrow$ {just predicate calculus... we're lucky if we get away with this strenghtening}

$p = [\,] \wedge s = P_\bullet +\!\!+ S_\bullet$

$\equiv$ $\triangleleft\, reset \,\triangleright$
{applying (5) gives a formally weaker single identity}

$p +\!\!+ s = P_\bullet +\!\!+ S_\bullet$

$\Leftarrow$ {separating initial values}

$p +\!\!+ s = P +\!\!+ S \wedge P = P_\bullet \wedge S = S_\bullet$

$\Leftarrow$ {definition of $A0$ and $A1$}

$A \wedge P = P_\bullet \wedge S = S_\bullet$,

which gives

```
  { $A$ }
  $reset$
; $\neg eof$
; $read(f)$
; $d, e, g := false, true, f$
; $P, S, Q, B; = [\,], [f] +\!\!+ s, in, f$
  { $R \wedge S \neq [\,] \wedge A$ } .
```

Combining this with the result for the other case gives

```
  { $A$ }
  $reset$
; if  $eof$ $\rightarrow$      $d, e := true, false$
                 ; $P, S, Q, B := [\,], [\,], in, \perp$
  [] $\neg eof$ $\rightarrow$   $read(f)$
                 ; $d, e, g := false, true, f$
                 ; $P, S, Q, B := [\,], [f] +\!\!+ s, in, f$
  fi
  { $R \wedge A$ } ,
```

in which no miraculous coercion statements occur any more. Finally, by omitting the abstract ghost variables, we get

```
  $reset$
; $\mathbf{if}$ $eof$ $\rightarrow$      $e := false$
  $[]$ $\neg eof$ $\rightarrow$   $read(f)$
                 ; $d, e, g := false, true, f$
  $\mathbf{fi}$ ,
```

and this is the code for *Reset* that our derivation produces.

Observing the code we have produced, it may strike us that variable e is inspected neither in *Reset* nor in any of the other operations that have not been reproduced here. Therefore e too can be regarded as a ghost variable and left out. With sufficient foresight, we might have seen this already in the specifications of the ISO operations: none of these require the definedness of the file buffer to be tested. From this observation, we might then have decided to replace $A2$ and $A3$ by the weaker

$$B = g \ \vee \ B = \perp \quad .$$

Leaving out e produces the final version of our implementation:

```
  $reset$
; $\mathbf{if}$ $eof$ $\rightarrow$      $skip$
  $[]$ $\neg eof$ $\rightarrow$   $read(f)$
                 ; $d, g := false, f$
  $\mathbf{fi}$
```

7 CONCLUSION

The extension we propose in this paper has been demonstrated to be practically useful in situations where either hypotheses or program statements are discovered one at a time as the calculation progresses, since it does away with the need to restart the calculation in a slightly different context every time such an ingredient is produced. Thus, it serves to make Dijkstra-style program derivation feasible for a wider class of programming problems.

References

Bijlsma, A. (1997) Simulating file operations: an exercise in calculational data refinement. Memorandum AB66, Eindhoven University of Technology, 1997. Available from `http://www.win.tue.nl/inf/staf/secties/st/pm/lexb/ab66.ps`

Bijlsma, A., Matthews, P.A., and Wiltink, J.G. (1989) A sharp proof rule for procedures in *wp* semantics. *Acta Inf.* **26**, 409–419.

Bijlsma, A. and Scholten, C.S. (1996) Point-free substitution. *Sci. Comput. Prog.* **27**, 205–214.

Dijkstra, E.W. and Feijen, W.H.J. (1984) *Een methode van programmeren.* Academic Service, The Hague. English translation (1988) *A method of programming.* Addison-Wesley, Reading (Mass.).

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

Gries, D. and Schneider, F.B. (1993) *A logical approach to discrete math.* Springer-Verlag, New York.

Grundy, J. (1996) Transformational hierarchical reasoning. *Comput. J.* **39**, 291–302.

Kaldewaij, A. (1990) *Programming: the derivation of algorithms.* Prentice-Hall International, London.

Morgan, C.C. (1990) *Programming from specifications.* Prentice-Hall International, London.

Morgan, C.C. and Gardiner, P.H.B. (1990) Data refinement by calculation. *Acta Inf.* **27**, 481–503.

Morris, J.M. (1989) Laws of data refinement. *Acta Inf.* **26**, 287–308.

8 BIOGRAPHY

A.J.M. van Gasteren received a master's degree in mathematics, supervised by Edsger W. Dijkstra.

In 1981 she joined Dijkstra's group as a BP Venture Research Fellow, investigating the orderly presentation and design of programs and proofs. As a result of this work, she received a PhD from Eindhoven University of Technology in 1988.

Since then she has taught and investigated programming methodology, getting more and more interested and involved in the use of calculational methods. A main driving force of her work is the quest for ways and methods to achieve clarity of thought and economy of expression.

A. Bijlsma studied number theory with H. Jager (Amsterdam), P.L. Cijsouw (Eindhoven), and M. Waldschmidt (Paris). In 1978 he received a PhD from the University of Amsterdam.

In 1983 he shifted his attention to computing. At Eindhoven University of Technology, he has since worked in the groups of M. Rem, A. Kaldewaij, and R.C. Backhouse. His research interest focuses on the interplay between programming language constructs and program construction style, in particular with respect to object-oriented design.