PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 2, PP. 77–99 (1991)

FACTORISATION OF THE FACTORIAL AN EXAMPLE OF INVERTING THE FLOW OF COMPUTATION[1]

E. A. BOITEN

Department of Informatics
Faculty of Mathematics and Informatics
University of Nijmegen, The Netherlands

Received: Sept. 5, 1990.

Abstract

As an example of the transformational programming method, a previously unknown algorithm for calculating factorials is derived. The derivation is done by the unfold-fold strategy with transformation rules for changing the recursion structure of functions. These transformation rules (inverting the flow of computation and splitting recursion) are presented and explained. The derivation proceeds from a system of linear recursive functions, via tail-recursive functions, to an efficient imperative program. The resulting program is, in our opinion, only intelligible by way of its derivation. It is also shown how a similar derivation leads to a version of the algorithm that may be executed on 2 processors.

Keywords: transformational programming, inversion of computation, factorial function.

1 Introduction

In the last decade, transformational programming has proved to be an appropriate methodology for developing programs (see, e.g., FEATHER, 1987; PARTSCH, 1990). The essence of this methodology is the derivation of (efficient) programs from formal specifications by applying *semantics preserving transformations,* i.e. applying a transformation rule results in a semantically equivalent program.

In a previous paper (BOITEN, 1989), we presented some transformation rules for *inverting the flow of computation.* Inverting the flow of computation is a transformation technique that can be applied to recursive functions, aiming at improvement of efficiency. The functions resulting from this transformation use (possibly, a.o.) the same arguments in the recursive evaluation as the original functions, but in an inverted order.

Later, we discovered a particular algebraic property of a (contrived) example function in (BOITEN, 1989). This led, via a sequence of transfor-

[1]This research has been sponsored by the Netherlands Organisation for Scientific Research (NWO), under grant NF 63/62–518 (the STOP — Specification and Transformation Of Programs — project).

mations (including inverting the flow of computation) to a, to our knowledge, new algorithm for computing factorials. It also appeared possible to derive a version of the algorithm for execution on 2 processors.

This paper is organised as follows. In the next two sections our framework is introduced: the methodology, the language, and some notations specific to this paper in Section 2, and a short description of the methodology in Section 3.

In Section 4, some transformation rules (*inverting the flow of computation and splitting linear recursion*) that are needed later are given.

The definition of *fact* using the new function *facthalf* is presented in Subsection 4.2. It is shown in Subsection 5.1 how a particular property can be used to optimise the new algorithm for the factorial function. Subsection 5.2 shows how the algorithm in Subsection 5.1 can be optimised by inverting the flow of computation. The efficiency of the resulting algorithm may be clearer to a computer scientist than it is to a mathematician, because multiplications and divisions by 2 are not considered 'special' in mathematics. The resulting algorithm is, in our opinion, only intelligible by way of its derivation.

A derivation of a variant of the algorithm in Subsection 5.2 that might be implemented to run on 2 processors is presented in Section 6.

A more extensive report on our manipulations of the function *facthalf* can be found in (BOITEN, 1990).

2 Language and Notation

In this Section we introduce some notation and present the language used in this paper.

A functional language is used that is similar to CIP-L (BAUER et al, 1985). Most of the constructs used here are self-explanatory. As in CIP-L, the semantics is strict and call-by-value. Because most functions in this paper are functions on natural numbers, the type **nat** of arguments and results is frequently omitted.

Many functions considered here are of the form:

$$(2.1)\quad f(x : Q(x)) = \begin{array}{l} \textbf{if } T(x) \\ \textbf{then } H(x) \\ \textbf{else } x \oplus f(K(x))\textbf{fi} \end{array}$$

The predicate $Q(x)$ restricts the domain of f to those elements that satisfy Q. $T(x)$, $H(x)$, and $K(x)$ are expressions in which the variable x may occur free, and in which the function f does not occur. $\oplus$ and $\otimes$ (used later) denote binary operators.

The notation $f^n(x)$, $n \geq 0$, denotes the n-fold application of f to x, i.e.:

$$f^0(x) = x\,,$$
$$f^n(x) = f(f^{n-1}(x))\,\text{for}\,n \geq 1\,.$$

The function g^{-1} denotes the inverse of g, provided that it exists. $(g^{-1})^k(x)$ is abbreviated to $g^{-k}(x)$.

3 Methodology

A derivation in the *transformational programming* methodology is presented. The essence of this methodology is the derivation of (efficient) programs from formal specifications by applying *semantics preserving transformations*, i.e. applying a transformation rule results in a semantically equivalent program.

The strategy we use is mainly the unfold-fold strategy (BURSTALL and DARLINGTON, 1977). *Unfolding* is the substitution of a function call by the body of the function, with substitution of the formal parameters by the actual parameters. *Folding* is the inverse of unfolding, i.e., an instance of a function body is replaced by a function call with suitable parameters.

Most phases of the derivation start with the introduction of a new function, defined in terms of existing ones. Some motivation is usually given for the introduction of the new function, we refer to well-known strategies like *finite differencing* (PAIGE and KOENIG, 1982) and *accumulation* (BIRD, 1984). Function calls are unfolded, often simplifications and rearrangements are done, until by folding an independent version of the new function can be obtained.

In this paper, some special transformation rules will be used as well (cf. Section 4). These are rules that require more complicated inductive proofs than can be provided by unfolding and folding only.

4 Transformation Rules

In this Section, we present three transformation rules to be used in the rest of this paper. The first two invert the flow of computation of functions of the form (2.1).

4.1 Inverting the Flow of Computation

We aim at transforming a function f of the form

```
(4.1)   $f(x : Q(x))$  if $T(x)$
                       then $H(x)$
                       else $x \oplus f(K(x))$fi
```

into an equivalent function of the form:

```
(4.2)   $f(x : Q(x))$                   $= f'(c, x)$
                                        where
        $f'(y, z : Q(y) \wedge Q(z))$   $=$ if $y = z$
                                          then $H(c)$
                                          else $K^{-1}(y) \otimes f'(K^{-1}(y), z)$fi
```

where K^{-1} is the inverse of K, and c is some fixed value, viz. the last argument to f in the original computation. Intuitively, this transformation rule transforms the calculation of a term

$$x_1 \oplus (x_2 \oplus (x_3 \oplus \ldots \oplus x_p) \ldots)$$

into the calculation of a term

$$x_{p-1} \otimes (x_{p-2} \otimes \ldots \otimes (x_2 \otimes (x_1 \otimes x_p) \ldots).$$

Thus, a computational sequence is inverted.

As an example of a function of the form (4.1), consider the well-known factorial function, defined by:

(4.3) $fact(x : x \geq 0) =$ **if** $x = 0$ **then** 1 **else** $x \times fact(x-1)$ **fi**

An important notion for inverting the flow of computation is the dependency relation between function calls. We say that argument x depends on argument y for function f, denoted by $x \leftarrow_f y$, if the value $f(y)$ is evaluated in order to determine the value of $f(x)$. Although this description suffices for our purposes, a formal definition is given in order to show that these dependency relations can be defined operationally, and can therefore be included in the program text.

Definition 4.1 $x \leftarrow_f y \equiv Q(x) \wedge (x = y \vee (\neg T(x) \wedge K(x) \leftarrow_f y))$ *(recall that f is of the form in program* 4.1*).*

Lemma 4.1

$$x \leftarrow_f y \equiv Q(x) \land \exists k \geq 0 : (y = K^k(x) \land \forall i : 0 \leq i < k : \neg T(K^i(x)))$$

Proof. Follows directly from definition 4.1. □

Often, computationally more efficient expressions for the dependency relation $\leftarrow_f$ can be derived. As an example, the following holds according to definition 4.1:

$$x \leftarrow_{fact} y \equiv x \geq 0 \land (x = y \lor (x \neq 0 \land x - 1 \leftarrow_{fact} y)),$$

where $fact$ is as defined above.

It can even be simplified to the following non-recursive expression:

$$x \leftarrow_{fact} y \equiv 0 \leq y \leq x\,.$$

As shown in Section 4, the simplification of such expressions for function call dependencies may be an important step in inverting the flow of computation.

The computational sequence of the original factorial function, viz.

$$\begin{tikzcd} \bullet \arrow[d] & fact(x) \\ \bullet \arrow[d] & fact(x-1) \\ \bullet \arrow[d] & fact(x-2) \\ \cdots & \end{tikzcd}$$

is transformed by transformation rule 4.3 into

$$\begin{tikzcd}
\bullet \arrow[d] & fact(x) \\
\bullet \arrow[d] & fact'(0,x) \\
\bullet \arrow[d] & fact'(1,x) \\
\bullet \arrow[d] & \ldots \\
\bullet & fact'(x,x) \\
\ldots &
\end{tikzcd}$$

More general conditions for this transformation rule can be found in (BOITEN, 1989). If we restrict ourselves to the most simple case, viz. where $\otimes \equiv \oplus$, we have:

Transformation 4.1

Defined and deterministic functions f of the form

$$(4.4)\quad f(x : Q(x)) = \begin{array}{l} \textbf{if } T(x) \\ \textbf{then } H(x) \\ \textbf{else } x \oplus f(K(x)) \textbf{ fi} \end{array}$$

are equivalent to

$$(4.5)\quad \begin{array}{ll} f(x : Q(x)) & = f'(c,x) \\ & \textbf{where} \\ f'(y,z : Q(y) \wedge Q(z)) & = \textbf{if } y = z \\ & \quad \textbf{then } H(c) \\ & \quad \textbf{else } K^{-1}(y) \oplus f'(K^{-1}(y), z) \textbf{ fi} \end{array}$$

under the following conditions:

1. *K is invertible, i.e. a function K^{-1} exists that fullfils*

$$\forall x : \neg T(x) \Rightarrow K^{-1}(K(x)) = x\,;$$

2. *$\oplus$ is left-commutative*

$$\forall x, y, z : x \oplus (y \oplus z) \equiv y \oplus (x \oplus z) \,;$$

3. every function call $f(x)$ depends on $f(c)$:

$$T(c) \land \forall x : Q(x) : x \leftarrow_f c \,.$$

In particular, when

$$T(x) \equiv x = q \,,$$

then we have $c = q$ in the transformed f.

So far, we have dealt only with functions for which a left inverse of K exists. For the inversion of computation it is, however, not necessary at all that the left inverse of K exists.

In the inverted version above, after determining the value of $f(y)$, $K^{-1}(y)$ is used to determine the value z such that $K(z) = y$. Note that this always takes place in the context of the inverted computation of $f(x)$ for a certain x. Although there may be multiple values z such that $K(z) = y$, only one of those values actually occurs as an argument of f in the computation of $f(x)$. PATERSON and HEWITT (1970) have shown that such a z can always be found, albeit in an inefficient way. This is done by reconstructing all arguments $K^i(x)$ for $i = 1 \ldots j$ such that $K^j(x) = y$. Such a j always exists, because y occurs as an argument to f in the computation of $f(x)$. The argument that precedes the argument y in the computation of $f(x)$ is $K^{j-1}(x)$. Formally, this way of finding in the context of the computation of $f(x)$ a z such that $K(z) = y$ can be defined by:

$$\begin{array}{lcl} K^{-1}_{PatHew}(x, y : \exists j \geq 1 : y = K^j(x)) & = & \textbf{if } y = K(x) \\ & & \textbf{then } x \\ & & \textbf{else } K^{-1}_{PatHew}(K(x), y) \textbf{ fi} \end{array}$$

We still use the symbol K^{-1}, because it denotes the inverse relation of K, restricted to the set $\{y | x \leftarrow_f y\}$. This will be called a *generalised left inverse* function.

A transformation rule similar to 4.1 can be given which uses the generalised inverse. The most important difference with the previous transformation rule is the incorporation of the argument x in the generalised left inverse K^{-1}. Also, the requirement that the starting value c is independent of the argument x has been dropped.

Transformation 4.2

Defined and deterministic functions f of the form

$$(4.6)\quad f(x : Q(x)) = \begin{array}{l}\textbf{if } T(x)\\ \textbf{then } H(x)\\ \textbf{else } x \oplus f(K(x)) \textbf{ fi}\end{array}$$

are equivalent to

$$(4.7)\quad \begin{array}{ll} f(x : Q(x)) & = f'(c)\\ & \textbf{where}\\ c & = \textbf{that } k : T(k) \wedge x \leftarrow_f k\\ f'(y : Q(y)) & = \textbf{if } y = x\\ & \quad \textbf{then } H(c)\\ & \quad \textbf{else } K^{-1}(x,y) \oplus f'(K^{-1}(x,y)) \textbf{ fi}\end{array}$$

under the following conditions:

1. *K^{-1} is the generalised inverse of K, i.e.*

$$\forall x, y : (x \leftarrow_f y \wedge \neg T(y)) \Rightarrow K^{-1}(x, K(y)) = y\ ;$$

2. *$\oplus$ is left-commutative*

$$\forall x, y, z : x \oplus (y \oplus z) \equiv y \oplus (x \oplus z)\ .$$

A useful lemma for finding more efficient generalised left inverses is the following:

Lemma 4.2

$$K^{-1}_{PatHew}(x, y : \exists j \geq 1 : y = K^j(x)) \equiv \textbf{that } z : \neg T(z) \wedge K(z) = y \wedge x \leftarrow_f z$$

4.2 Splitting Linear Recursion

In (BOITEN, 1989), the following alternative definition of the factorial function was presented:

$$(4.8)\quad fact(x) \;=\; \begin{array}{l}\textbf{if } x = 0\\ \textbf{then } 1\end{array}$$

```
                    else $facthalf(x) \times facthalf(x-1)$ fi
    where
    $facthalf(x)$   = if $x \le 1$
                      then 1
                      else $x \times facthalf(x-2)$ fi
```

It is clear that the above function calculates the factorial of a number by calculating the products of the odd and even factors separately. Intuitively, its correctness is obvious. Formally, it is guaranteed by the correctness of the following transformation rule, which is proved in (BOITEN, 1990):

Transformation 4.3

Defined functions f of the form

```
(4.9)   $f(x) =$  if $T(x)$
                  then $H(x)$
                  else $Q(x) \oplus f(K(x))$ fi
```

are equivalent to

```
(4.10)   $f(x)$  =  if $T(x)$
                    then $\bigoplus_{j=0}^{-1} Q(K^j(x)) \oplus H(x)$
                    elsf $T(K(x))$
                    then $\bigoplus_{j=0}^{0} Q(K^j(x)) \oplus H(K(x))$
                    ...
                    elsf $T(K^{n-1}(x))$
                    then $\bigoplus_{j=0}^{n-2} Q(K^j(x)) \oplus H(K^{n-1}(x))$
                    else $\bigoplus_{i=0}^{n-1} fn(K^i(x))$
                    fi
          where
          $fn(x)$  =  if $T(x)$ then $H(x)$
                    elsf $\exists_{i=1}^{n-1} T(K^i(x))$
                    then $Q(x)$
                    else $Q(x) \oplus fn(K^n(x))$
                    fi
```

for any $n \ge 1$, provided that the operator $\oplus$ is associative and commutative, with unit element $1_\oplus$. The expressions $\bigoplus_{i=p}^{q} g(i)$ are defined by:

$$\bigoplus_{i=p}^{q} g(i) = \begin{cases} 1_\oplus & if\, p > q\,, \\ g(p) \oplus \left(\bigoplus_{i=p+1}^{q} g(i)\right) & otherwise\,. \end{cases}$$

Note that the only expression of the form $\bigoplus_{i=p}^{q} g(i)$ that does not contain a term syntactically different from $1_\oplus$ occurs only in the context $\bigoplus_{i=p}^{q} g(i) \oplus H(x)$. That means that, for fixed n, all expressions of the form $\bigoplus_{i=p}^{q} g(i)$ in the transformation rule may be eliminated or simplified to expressions not containing $1_\oplus$. So, the value $1_\oplus$ may be fictitious, i.e. if no unit of $\oplus$ exists, a new element $1_\oplus$ may be adjoined to the type of $\oplus$'s operands. The only property that is required of the new value $1_\oplus$ is that for all **t** $x : x \oplus 1_\oplus = x$.

Intuitively, this transformation rule splits the calculation of a term

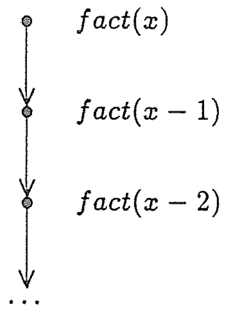
$$x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$$

into n calculations of terms

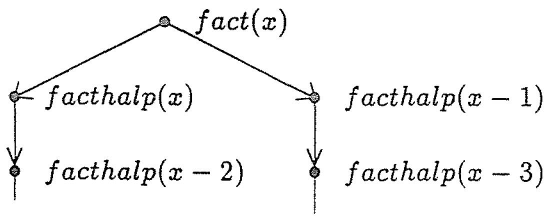
$$x_1 \oplus x_{n+1} \oplus x_{2n+1} \oplus \ldots ,$$
$$x_2 \oplus x_{n+2} \oplus x_{2n+2} \oplus \ldots ,$$
$$\ldots$$
$$x_n \oplus x_{2n} \oplus x_{3n} \oplus \ldots .$$

Thus, a computational sequence is transformed into a computational tree with n branches.

The computational sequence of the original factorial function, viz.



is transformed by transformation rule 4.3 into



(for $x \geq 3$). Thus, one computation is split up into two independent computations that may be executed in parallel. It is clear that transformation rule 4.3 may be used for the evaluation of certain kinds of linear recursive functions on architectures with a fixed number (≥ 2) of processors.

5 Transformational Development

5.1 A Property of facthalf

By induction, it can easily be proved that the following property holds for the function *facthalf* in program (4.8).

$$even(x) \Rightarrow facthalf(x) = 2^{x/2} \times fact(x/2) . \qquad (1)$$

This property is also mentioned in (PURDON and BROWN, 1985, exercise 4 on page 98). In the following, / denotes integer division.

```
(5.1)                     $facthalf(x) \times facthalf(x-1)$
        $=_{\{(1)\}}$             **if** $odd(x)$
                          **then** $facthalf(x) \times 2^{(x-1)/2} \times fact((x-1)/2)$
                          **else** $fact(x/2) \times 2^{x/2} \times facthalf(x-1)$
                          **fi**
        $=_{\{\text{use } \mathbf{mod}, \text{comm.}\times\}}$  $facthalf(x - 1 + x\,\mathbf{mod}\,2) \times 2^{x/2} \times fact(x/2)$
```

Finally, it is clear that all arguments to *facthalf* are odd. When we add this to the assertion in *facthalf*, and simplify *facthalf* accordingly, we have altogether:

```
(5.2)  $fact(x)$
       $=$  **if** $x = 0$ **then** 1
            **else** $facthalf(x - 1 + x\,\mathbf{mod}\ 2) \times 2^{x/2} \times fact(x/2)$
            **fi**
        **where**
       $facthalf(x : x \geq 1 \wedge odd(x))$
       $=$  **if** $x = 1$
            **then** 1
            **else** $x \times facthalf(x-2)$**fi**
```

This is our second, in our view surprising version of *fact*, which uses mainly subtraction and division by 2 instead of subtraction by 1 in its recursion.

5.2 Possibilities for improving *fact*/*facthalf*

As an example, consider the evaluation of $fact(31)$. By repeatedly unfolding $fact$, we get:

$$\begin{aligned} fact(31) = & facthalf(31) \times facthalf(15) \times facthalf(7) \times \\ & facthalf(3) \times facthalf(1) \times 2^{26} . \end{aligned}$$

Obviously,

$$facthalf(31) = 31 \times 29 \times \ldots \times 17 \times facthalf(15) ,$$

i.e., there is some redundancy in the computation. We are, however, not able as yet to eliminate that redundancy. This is because $facthalf(31)$ is computed 'first', and only later is the intermediate result $facthalf(15)$ again useful. Therefore, we aim at inverting the flow of computation of $fact$. The result of that is that $fact(15)$ is computed first in the computation of $fact(31)$, and it is multiplied only afterwards with the value of $facthalf(31)$. Thus, $facthalf(15)$ is used before $facthalf(31)$, and can be used as a starting value for computing $facthalf(31)$. If we want to do so, we have to find a version of $facthalf$ that computes as $facthalf(31)$ as $(\ldots((facthalf(15) \times 17) \times 19) \times \ldots) \times 31$. This can be achieved by also inverting the flow of computation of $facthalf$.

5.3 Transforming *facthalf*

Transformation rule 4.1 can be applied to $facthalf$, resulting in:

```
(5.3)  $facthalf(x : x \geq 1 \wedge odd(x))$
       $= fh(1, x)$
        **where**
       $fh(y, x : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
       $=$   **if** $y = x$
             **then** $1$
             **else** $(y + 2) \times fh(y + 2, x)$ **fi**
```

The correctness of this version is guaranteed by the invertibility of $K(x) = x - 2$ and the commutativity of multiplication.

We now prove a lemma that will be useful later in the derivation.

Lemma 5.1
For all p, q, x such that $1 \le p \le q \le x \wedge odd(p) \wedge odd(q) \wedge odd(x)$,
$fh(p,x) = fh(p,q) \times fh(q,x)$.

Proof: Intuitively it is clear that the lemma holds, because $fh(p,q)$ is simply the product of all odd numbers from $p \# 2$ up to q. Formally, this can be proved by induction on $q-x$.

Basis. If $q-x=0$, then $fh(p,x) = fh(p,x) \times 1 = fh(p,q) \times fh(q,x)$.

Induction. Suppose the lemma holds for $x-2k \le q \le x$. Then

$$\begin{aligned}
&fh(p, x-2k-2) \times fh(x-2k-2, x) =_{\{\text{unfold}\, fh\}}\\
&fh(p, x-2k-2) \times (x-2k) \times fh(x-2k, x) =_{\{\text{fold}\, fh,\, \text{commutativity} \times\}}\\
&fh(p, x-2k) \times fh(x-2k, x) =_{\{\text{induction}\}}\\
&fh(p,x). \quad \square
\end{aligned}$$

Because multiplication is associative, the accumulation strategy (BIRD, 1984) can be applied by definition of

$$fhf(x,y,res) = res \times fh(x,y) \tag{2}$$

resulting in:

```
(5.4)  $facthalf(x : x \ge 1 \wedge odd(x))$
       $=$  $fhf(1, x, 1)$
        where
       $fhf(y, x, res : x \ge y \ge 1 \wedge odd(x) \wedge odd(y))$
       $=$  if $y = x$
            then $res$
            else $fhf(y+2, x, res \times (y+2))$
            fi
```

The function fhf can be transformed into iterative form, because it is tail recursive. Then we have:

```
5.5  $fhf(y, x, res : x \ge y \ge 1 \wedge odd(x) \wedge odd(y))$
     $=$ begin var$(vy, vx, vres) := (y, x, res)$;
               while $vy \ne vx$
               do $(vy, vx, vres) := (vy+2, vx, vres \times (vy+2))$
               od;
               $vres$
         end
```

This can be simplified by eliminating all assignments to vx and replacing all other occurrences of vx by x, yielding:

```
5.6  $fhf(y, x, res : x \geq y \geq 1 \wedge odd(x) \wedge odd(y))$
     = begin var$(vy, vres) := (y, res)$;
             while $vy \neq vx$
             do $(vy, vres) := (vy + 2, vres \times (vy + 2))$
             od;
             $vres$
       end
```

5.4 Transforming fact

As mentioned before, in order to profit from the new version of *facthalf*, we need to invert the flow of computation of *fact* as well. Furthermore, some optimisations (viz. accumulation and finite differencing) are possible afterwards.

First the complicated expression $x-1+(x \,\mathbf{mod}\, 2)$ is abstracted. Note that it denotes the greatest odd number less than or equal to x. This definition of *toodd* is used throughout this paper.

(5.7) $\quad toodd(x : x > 0) = x - 1 + (x \,\mathbf{mod}\; 2)$

The function *fact* can now be improved by inverting the flow of computation. This will be done along the lines of Section 4.

If $K(x) = x/2$ were invertible, transformation rule 4.1 would be applicable. This is not the case, and thus we should find a generalised left inverse of K in order to apply transformation rule 4.2. Since $K^j(x) = x/2^j$, we can define K^{-1}, using lemma 4.2, by:

(5.8) $\quad K^{-1}(x, y : \exists k > 0 : y = x/2^k) = \mathbf{that}\ z : z/2 = y \wedge \exists k : z = x/2^k$

Later on, an efficient definition of K^{-1} can be given. The definition of K^{-1} will not be repeated in the derivations. Using transformation rule 4.2, we now invert the flow of computation of *fact*, resulting in:

```
(5.9)  $fact(x) = fact'(0)$
       where
       $fact'(y)$
       =   if $y = x$
           then 1
           else $2^y \times facthalf(toodd(ny)) \times fact'(ny)$
             where $ny = K^{-1}(x, y)$
           fi
```

By using the definition of *facthalf* above, the **else**-branch transforms into:

$$2^y \times fhf(1, toodd(ny), 1) \times fact'(ny).$$

The next goal is now improvement of *fact* by finite differencing. We aim at carrying along the value of *fhf* last computed. Furthermore, because the last call of *fhf* has *toodd*(*ny*) as an argument, the value of *toodd*(*ny*) will also be kept. First we define a new function *fact''* with appropriate assertion (note that when $y=0$, no *fhf* value has been computed yet, and thus the assertion should give no extra information):

```
(5.10)  $fact''(\ y, z, oddy :$
                $y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
        $= fact'(y)$
```

By unfolding, abstraction and simplification we get:

```
(5.11)  $fact''(\ y, z, oddy :$
                $y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
        $=$  if $y = x$
             then $1$
             else $2^y \times fhf(1, toodd(ny), 1 \times fact'(ny)$
               where $ny = K^{-1}(x, y)$
             fi
```

The following simplification is possible:

$$\begin{aligned}
&fhf(1, toodd(ny), 1)\\
&\qquad =_{\{\text{ lemma 5.1, def. } fhf\}} fhf(1, oddy, 1) \times fhf(oddy, toodd(ny), 1)\\
&\qquad =_{\{(2)\}} fhf(oddy, toodd(ny), fhf(1, oddy, 1))\\
&\qquad =_{\{\text{assertion } fact''\}} fhf(oddy, toodd(ny), z)
\end{aligned}$$

Using this, we can fold *fact''* (the *fhf* value just computed is the correct new value for *z*, according to the assertion), resulting in:

```
(5.12)  $fact''(y, z, oddy :$
                $y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
        $=$  if $y = x$
             then $1$
             else $2^y \times nz \times fact''(ny, nz, oddny)$
               where  $ny = K^{-1}(x, y), oddny = toodd(ny),$
                      $nz = fhf(oddy, oddny, z)$
             fi
```

For $fact$, we then have

$$fact(x) = fact''(0,1,1).$$

Due to commutativity of multiplication and addition, the accumulation strategy can also be applied to $fact''$. We define

$$fact'''(y, z, oddy, res, two) = fact''(y, z, oddy) \times res \times 2^{two}.$$

This allows the derivation of:

```
(5.13)  $fact(x)$
        $=$ $fact'''(0,1,1,1,0)$
          where
        $fact'''(y, z, oddy, res, two :$
              $y \neq 0 \Rightarrow (oddy = toodd(y) \wedge z = fhf(1, oddy, 1)))$
        $=$ if $y = x$
            then $res \times 2^{two}$
            else $fact'''(ny, nz, oddny, res \times nz, two + y)$
              where $ny = K^{-1}(x, y), oddny = toodd(ny),$
                    $nz = fhf(oddy, oddny, z)$
            fi
```

A parameter n is added, such that $y = x/2^n$. Because $ny = x/2^{n-1}$, we have a more efficient expression for K^{-1}. The initial value should be $\lfloor {}^2\log x \rfloor + 1$ for $x > 0$, since $x/2^{\lfloor {}^2\log x \rfloor + 1} = 0$. Because ${}^2\log 0$ is undefined, we single out 0 in the definition of $fact$, which results in:

```
(5.14)  $fact(x)$
        $=$ if $x = 0$ then 1
            else $fact'''(0,1,1,1,0)$ fi
```

Note that K^{-1} might be implemented even more efficiently: all first arguments to $fact'''$ are of the form $x/2^k$, and so their binary representations are *prefixes* of the binary representation of x. K^{-1} transforms a prefix of length n into a prefix of length $n+1$, and this could also be achieved by gradually *shifting* x into a location.

Finite differencing by introduction of a parameter such that $y = x/2^n$ yields:

```
(5.15)  $fact(x)$
        $=$ if $x = 0$ then 1
```

```
    else $fact''''(0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1)$ fi
  where
$fact''''(y, z, oddy, res, two, n$:
    $oddy = toodd(y) \wedge z = fhf(1, oddy, 1) \wedge y = x/2^n)$
= if $y = x$
    then $res \times 2^{two}$
    else $fact''''(ny, nz, oddny, res \times nz, y + two, n - 1)$
      where  $ny = x/2^{n-1}, oddny = toodd(ny)$,
             $nz = fhf(oddy, oddny, z)$
    fi
```

5.5 The Imperative Level

Because $fact''''$ is tail recursive, we can transform (5.15) (with unfolding of all value abstractions and the imperative counterpart of $fact''''$) into:

```
(5.16)  $fact(x)$
        = if $x = 0$ then 1 else
            begin
              var $(vy, vz, voddy, vres, vtwo, vn) := (0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1)$;
              while $vy \neq x$
              do
                 $(vy, vz, voddy, vres, vtwo, vn) :=$
                 $(x/2^{vn-1}, fhf(voddy, toodd(x/2^{vn-1}), vz), toodd(x/2^{vn-1})$
                 $, vres \times fhf(voddy, toodd(x/2^{vn-1}), vz), vtwo + vy, vn - 1)$
              od;
              $vres \times 2^{vtwo}$
            end fi
```

The inner assignment statement can be sequentialised as follows:

```
(5.17)  $vtwo := vtwo + vy$;
        $vn := vn - 1$;
        $vy := x/2^{vn}$;
        $vz := fhf(voddy, toodd(vy), vz)$;
        $voddy := toodd(vy)$;
        $vres := vres \times vz$
```

Now we unfold fhf in the assignment to vz, yielding:

```
(5.18)  $vz :=$  begin
                   var $(va, vb) := (voddy, vz)$;
```

```
    while $va \neq toodd(vy)$
    do $(va, vb) := (va + 2, vb \times (va + 2))$ od
    $vb$
  end;
```

The following optimisations are now possible:

- because *va* is initialised with *voddy*, and *voddy* is not used in the inner loop, and *va* equals *toodd*(*vy*) upon termination, *voddy* can replace *va*, thereby making the assignment to *voddy* superfluous;
- *vb* is initialised with *vz*, *vz* is not used in the inner loop, and after the inner loop *vz* is assigned *vb*; thus, *vz* can replace *vb*. This can also be derived via a sequence of small transformation steps.
- The assignments in the inner loop can be sequentialised in such a way that the expression *va*+2 (now: *voddy*+2) is computed only once.

This yields our final program, in which independent collateral assignments are not sequentialised:

```
(5.19)  $fact(x)$
        = if $x = 0$ then 1 else
            begin
              var $(vy, vz, voddy, vres, vtwo, vn) := (0, 1, 1, 1, 0, \lfloor {}^2\log x \rfloor + 1)$;
              while $vy \neq x$
              do    $vtwo := vtwo + vy$;
                    $vn := vn - 1$;
                    $vy := x/2vn$;
                    while $voddy \neq toodd(vy)$
                     do   $voddy := voddy + 2$;
                          $vz := vz \times voddy$
                    od;
                    $vres := vres \times vz$
              od;
              $vres \times 2^{vtwo}$
            end fi
```

6 Implementation on Two Processors

We will demonstrate how the above algorithm can be implemented on two processors. The first processor sends a sequence of appropriate *facthalf* values to the second one, which computes *fact* using those values.

In order to derive a version of the algorithm which is close to a parallel one, we need to introduce sequences. Often, in functional descriptions of parallel systems so-called streams are used to describe the communication (BROY and BAUER, 1984), but in this case only finite streams, i.e., sequences are needed. Because now multiple types occur in our functions, we write **nat** for natural number arguments and results, **bool** for booleans, and **seq** for sequences of natural numbers.

The type seq	
<>	the empty sequence
$\Vert$	prepend a natural number to a sequence
first	the first element of a sequence
rest	all but the first element

The derivation starts from the version of $fact$ in program (5.9) **nat** x is assumed to be known in the context. Below, a function $factp2$ is defined which is equal to $fact'$, except that it takes as an extra argument the sequence of all necessary $facthalf$ values. This is expressed by the predicate $allfh$.

```
(6.1)  $factp2(\mathbf{seq}\ s, \mathbf{nat}\ y : y \leq x \wedge allfh(s,y))\mathbf{nat}$
       $= fact'(y)$
       where
       $allfh(\mathbf{seq}\ s, \mathbf{nat}\ y)\mathbf{bool}$
       $=$ if $y = x$ then $s =<>$
       $=$ else first $s = facthalf(toodd(ny)) \wedge allfh(ny, \mathbf{rest}\ s)$
             where $ny = K^{-1}(x,y)$ fi
```

In order for $factp2$ to replace $fact'$, we need to derive:

- a definition of $factp2$ independent of $fact'$, and
- a value z such that $factp2(z,0) = fact'(0)$; in particular, this means that the assertion $allfh(z,0)$ should hold.

First we derive a definition of $factp2$.

```
$factp2(s,y) =_{\{\text{unfold } fact'\}}$   if $y = x$ then 1
                                  else $2^y \times facthalf(toodd(ny)) \times fact'(ny)$
                                  where $ny = K^{-1}(x,y)$ fi
           $=_{\{\text{definition } allfh\}}$   if $y = x$ then 1
                                  else $2^y \times$ first $s \times factp2(\mathbf{rest}\ s, ny)$
                                  where $ny = K^{-1}(x,y)$ fi
```

The value of s for the initial call can be computed from the assertion. A value z is needed, such that $allfh(z,0)$ holds. That value of z will be denoted by $factp1$, which implicitly depends on x, but also on y. The dependence on y is necessary to derive a recursive definition of $factp1$. In the derivation below, ny is assumed to be $K^{-1}(x, y)$.

```
factp1(y) ={above}
                some s : allfh(s, y)
          ={def. allfh}
                some s : if y = x then s =<>
                              else first  s = facthalf(toodd(ny))
                                   ∧allfh(rest s, ny)
                            fi
          ={distributivity}
                 if  y = x
                 then some s : s =<>
                 else some s : first  s = facthalf(toodd(ny))
                                    ∧allfh(rest s, ny)  fi
          ={some-simplification}
                 if  y = x
                 then  <>
                 else some s : first  s = facthalf(toodd(ny))
                                    ∧ allfh(rest s, ny)  fi
          ={seq. decomposition}
                 if  y = x
                 then  <>
                 else  facthalf(toodd(ny))
                          ⧺some  s' : allfh(s', ny) fi
          ={fold  factp1}
                 if  y = x
                 then  <>
                 else  facthalf(toodd(ny)) ⧺ factp1(ny)  fi
```

Then we have altogether:

```
(6.2)  fact(nat x)nat
       = factp2(factp1(0), 0)
         where
       factp1(nat y)seq
       = if  y = x
         then  <>
         else  facthalf(toodd(ny))⧺factp1(ny)
          where  ny = K⁻¹(x, y)  fi,
       factp2(seq  s,nat y)nat
```

```
    = if  $y = x$  then  1
      else  $2^y \times$  first  $s \times factp2($rest  $s, ny)$
      where  $ny = K^{-1}(x, y)$  fi,
    $facthalf($nat $y)$nat
    = if  $y = 1$  then  1
      else  $y \times facthalf(y-2)$  fi
```

The functions *factp*1 and *factp*2 can be optimised similarly to *fact* and *facthalf* in the previous section:

Optimise *factp*1:

- Define (finite differencing)

$$factp1(y) = factp1'(y, 1),$$
$$factp1'(y, z : y \neq 0 \Rightarrow z = facthalf(toodd(y))) = factp1(y)$$

- Use lemma 5.1 to derive

```
(6.3)  $factp1'(y, z)$
       = if  $y = x$  then  $<>$
         else  $nz +\!\!+ factp1'(ny, nz)$
               where  $ny = K^{-1}(x, y)$,
                      $nz = fhf(toodd(y), toodd(ny), z)$
         fi
```

Optimise *factp*2:

- Define (accumulation)

$$factp2(s, y) = factp2'(s, y, 0, 1)$$
$$factp2'(s, y, two, res) = 2^{two} \times res \times factp2(s, y)$$

- Derive

```
(6.4)  $factp2'(s, y, two, res)$
       = if  $y = x$
         then  $2^{two} \times res$
         else  $factp2'($rest  $s, ny, two + y, res \times$ first  $s)$
          where  $ny = K^{-1}(x, y)$  fi
         fi
```

- Implement K^{-1} as in the previous section.

It is clear that the sequence s in *factp*1 and *factp*2 may also be viewed as a one-way communication channel. Cf. (Broy and Bauer, 1984) for a discussion of this kind of consumer-producer programs. The function *factp*1 may be implemented on one processor, and send the computed successive first elements of s to the other processor on which *factp*2 is implemented.

It is even possible to use a third processor for computing the sequence of K^{-1} values. In the current version, these are computed by both *factp*1 and *factp*2.

7 Conclusions

We have given transformation rules that split and invert computation sequences of linear recursive functions.

For the factorial function, the application of these rules and subsequent manipulations resulted in a previously unknown algorithm. The resulting algorithm appears very complicated, and is in our opinion only intelligible by way of its derivation.

In (BORWEIN, 1985) a factorial algorithm is presented which is based on factoring out *all* prime factors. Its time complexity is better than that of our algorithm. However, it needs more space, viz. for a table of all prime numbers up to the argument of *fact*.

Finally, it was shown that the resulting algorithm may efficiently be executed on a two processor architecture.

Acknowledgement

I thank Helmut Partsch for pointing out the possibility of deriving a new algorithm from the first version of *fact/facthalf* and for much constructive criticism. Furthermore, Norbert Völker and Daniël Tuijnman are thanked for their remarks on earlier versions of this paper.

References

BAUER, F. L. – BERGHAMMER, R. – BROY, M. – DOSCH, W. – GEISELBRECHTINGER, F. – GNATZ, R. – HANGEL, E. – HESSE, W. – KRIEG-BRÜCKNER, B. – LAUT, A. – MATZNER, T. – MÖLLER, B. – NICKL, F. – PARTSCH, H. – PEPPER, P. – SAMELSON, K. – WIRSING, M. – WÖSSNER, H. (1985): The Munich Project CIP. Volume I: *The Wide Spectrum Language CIP-L, Lecture Notes in Computer Science* 183. Springer-Verlag, Berlin/Heidelberg/New York.

BIRD, R. S. (1984): The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 4, pp. 487–504.

BOITEN, E. A. (1989): Inverting the Flow of Computation. *Technical Report 89-10,* Dept. of Informatics, K.U. Nijmegen. To appear in Science of Computer Programming.

BOITEN, E. A. (1990): Factorisation of the Factorial – an Algorithm Discovered by Playing with Transformations. *Technical Report 89-10,* Dept. of Informatics, K.U. Nijmegen.

BORWEIN, P. B. (1985): On the Complexity of Calculating Factorials. *Journal of Algorithms*, Vol. 6, pp. 376–380.

BROY, M. – BAUER, F. L. (1984): A Systematic Approach to Language Constructs for Concurrent Programs. *Science of Computer Programming,* Vol. 4, pp 103–139.

BURSTALL, R. M. – DARLINGTON, J. (1977): A Transformation System for Developing Recursive Programs. *Journal of the ACM,* Vol. 24, No. 1, pp. 44–67.

FEATHER, M. S. (1987): A Survey and Classification of some Program Transformation Approaches and Techniques. In L.G.L.T Meertens, editor, *Program Specification and Transformation. Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation,* pp. 165–196. North-Holland Publishing Company, Amsterdam.

PAIGE, R. – KOENIG, S. (1982): Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 3, pp. 402–454.

PARTSCH, H. (1990): Specification and Transformation of Programs - a Formal Approach to Software Development. Springer-Verlag, Berlin.

PATERSON, M. S. – HEWITT, C. E. (1970): Comparative Schematology. *Record of the Project MAC Conf. on Conc. Syst. and Par. Comp.,* Woods Hole, Mass., pp. 119–127. ACM, New York.

PURDON, C. A. JR. – BROWN, C. A. (1985): The Analysis of Algorithms. Holt, Rineheart and Winston, New York.

Address:

Eerke A. BOITEN
Department of Informatics
Faculty of Mathematics & Informatics
University of Nijmegen
Toernooiveld 1
NL – 6525 ED Nijmegen
The Netherlands