The many disguises of accumulation

Eerke Boiten

University of Nijmegen, Dept. of Informatics
Technical Report 91-26, December 1991

Abstract

Several descriptions of basically one transformation technique, viz. *accumulation*, are compared. Their basis, viz. the associativity and the existence of a neutral element inherent in a monoid, is identified.

Keywords

transformational programming, factorial, fast reverse, accumulation, continuations, lambda abstraction, generalisation, tail recursion, implementation of lists.

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).

1 Introduction

One of the first program transformations that appeared in the literature was the *accumulation* transformation. The transformation is now classic, although not everyone may know it under exactly this name.

In this note, I try to relate several descriptions of this program transformation technique. In a purely algebraic view, it is the exploitation of the properties of a monoid. In literature, it can be found under the headings of *recursion removal* [Str71, WS73, AS78] or - *simplification* [Par90], *generalisation of output terms* [Aub75], *data structures representing continuations* [Wan80], *rebracketing* [BW82], *accumulation* [Bir84a], *lambda abstraction* or *higher order generalisation* [Pet87, PS87], or even *a novel implementation of lists* [Hug86]. It falls under the *generalisation tactic* in Feather's survey paper [Fea87], but is not mentioned explicitly there.

I will start by showing the two "famous examples" of accumulation. Then I show the basis of the technique with the monoid. Finally, I comment on various papers by showing how they describe accumulation and what other transformations are possible using the described technique.

The examples come from many sources, each with their own variant of a functional notation, which is in some cases the clearest notation to express their ideas in. Therefore, I ask to be excused for the variations in notation throughout this note.

Both juxtaposition and bracketing are used for function application; $\circ$ denotes function composition; $\oplus$ always denotes a binary infix operator, with unit element $1_\oplus$.

2 Famous example # 1: reverting lists

Assuming we are dealing with *cons*-lists over a (here, irrelevant) type α, defined by:

$$\frac{}{[\,] : \mathit{List}(\alpha)} \qquad\qquad \frac{x : \alpha, l : \mathit{List}(\alpha)}{(x + l) : \mathit{List}(\alpha)}$$

and, additionally, a function $+\!\!+$ (append) defined by:

$$\begin{aligned} [\,] +\!\!+ l &= l \\ (a + l_1) +\!\!+ l_2 &= a + (l_1 +\!\!+ l_2) \end{aligned}$$

i.e., the use of $l_1 +\!\!+ l_2$ forces iteration over l_1. Thus, the reverse of a list can be defined by:

$$\begin{array}{rcl} rev[\,] & = & [\,] \\ rev(a+l) & = & (rev\ l) +\!\!\!+ (a+[\,]) \end{array}$$

which is (given this definition of $+\!\!\!+$) an algorithm quadratic in the length of the argument. By some, as of yet unknown, motive we are led to define

$$h\ l_1\ l_2 \quad = \quad (rev\ l_1) +\!\!\!+ l_2.$$

Obviously, then

$$rev\ l \quad = \quad h\ l[\,]$$

and also

$$\begin{array}{rcl} h[\,]l_2 & = & l_2 \\ h(a+l_1)l_2 & = & h\ l_1(a+l_2) \end{array}$$

which constitutes a linear algorithm for rev.

3 Famous example # 2: accumulator version of factorial

We are given the usual definition of the factorial function *fact*, i.e.

$$\begin{array}{rcll} fact(0) & = & 1 & \\ fact(n) & = & n \times fact(n-1) & \text{for } n \geq 1 \end{array}$$

By generalising $fact(n)$ to $k \times fact(n)$ we obtain:

$$fact'(k,n) \quad = \quad k \times fact(n)$$

and thus,

$$fact(n) \quad = \quad fact'(1,n)$$

and also

$$\begin{array}{rcl} fact'(k,0) & = & k \\ fact'(k,n) & = & fact'(k \times n, n-1) \end{array}$$

which is a tail-recursive version of $fact$.

4 The monoid

According to [Gil76], a binary algebra $\mathcal{S} = <S; \oplus>$ is a monoid iff $\oplus$ is associative, i.e. for all $a, b, c \in S$:

$$a \oplus (b \oplus c) \quad = \quad (a \oplus b) \oplus c$$

and an element $1_\oplus \in S$ exists, such that, for all $a \in S$,

$$\begin{aligned} 1_\oplus \oplus a &= a \\ a \oplus 1_\oplus &= a \end{aligned}$$

In the examples, we encountered two monoids: $<List; +\!\!+>$, and $<\mathbf{Nat}; \times>$, with unit elements [ ] and 1, respectively.

The functions in the examples are of the form:

$$\begin{aligned} f(x) \quad = \quad & \textbf{if } T(x) \\ & \textbf{then } 1_\oplus{}^{1} \\ & \textbf{else } f(K(x)) \oplus H(x) \\ & \textbf{fi} \end{aligned}$$

so they compute

$$1_\oplus \oplus (x_1 \oplus (x_2 \oplus (\ldots) \ldots))$$

with $x_i = K^i(x)$ ($i = 1 \ldots n$ for some n such that $T(K^n(x))$), which is equal to (by associativity)

$$(\ldots((1_\oplus \oplus x_1) \oplus x_2) \oplus \ldots)$$

so, an equivalent definition is

$$\begin{aligned} f(x) \quad &= \quad f'(x, 1_\oplus) \\ f'(x, y) \quad &= \quad \textbf{if } T(x) \\ & \qquad \textbf{then } y \\ & \qquad \textbf{else } f'(K(x), H(x) \oplus y) \\ & \qquad \textbf{fi.} \end{aligned}$$

The crucial step in a unfold-fold style derivation is the definition of f' in terms of f:

$$f'(x, y) = f(x) \oplus y$$

(the *generalisation* step). This transformation was first presented in a comparable form by Darlington and Burstall [Dar72, DB76].

[1] This is not essential; when $T(x)$, $f(x)$ may also be $G(x)$ for the derivation to be possible. In that case, $f'(x, y)$ will be $G(x) \oplus y$ when $T(x)$.

5 A taxonomy

In the literature, roughly speaking three strategies that allow the above derivation can be discerned.

The simple *accumulation strategy* allows only the derivation of accumulator versions of functions, as in the above examples.

Embedding of the domain (a parameter is added to a function) is the underlying strategy for accumulation, but also for other strategies like *finite differencing* [PK82].

Continuation-based transformation strategies also allow the derivation of programs with functions as parameters.

All three strategies rely on the generalisation of a constant to a parameter; the constant may not always be visible, however.

6 On *the equivalence of certain computations*

In the early sixties, the advent of ALGOL, and McCarthy laying the foundation for a mathematical theory of computation [McC60], paved the way for the formal study of programs and algorithms.

The first paper I found that deals with accumulation is [Coo66]. There, one can already find the classical examples: factorial and fast reverse. By generalising those functions and the properties that guarantee the equivalence of their different versions, Cooper derives a general theorem for proving the equivalence of certain computations.

Cooper also gives the first formal treatment of *inverting the order of evaluation* [Boi92] for the factorial function.

7 Some descriptions by R. Bird and others

In [Bir84a], the use of the *accumulation strategy* in transformational programming is described. Bird considers *parameter accumulation* to be one of the most important things to be taught in functional programming. In his notation, the conditions for applicability of parameter accumulation are given as follows:

- Given a specification of the form

$$\mathit{spec}\ x = f(H\ x)$$

where $H\ x$ is a "large" object, defined by (when x is a list)

$$\begin{aligned} H[\,] &= \ldots \\ H(a;x) &= h\ a\ (H\ x) \end{aligned}$$

and a function f' exists such that

$$f\ x = f'\ c\ x$$

for some constant c;

- and the following condition holds:

$$(f'\ s) \circ (h\ a) \quad = \quad (h'\ a) \circ (f'(g\ a\ s))$$

for suitable functions g and h',

- then *spec* can be generalised to a function

$$spec'\ s\ x = f'\ s\ (H\ x),$$

- and we get

$$\begin{aligned} spec\ x &= spec'\ c\ x \\ spec'\ s(a;x) &= h'\ a(spec(g\ a\ s)x). \end{aligned}$$

Lately, R. Bird has been developing a formalism with L. Meertens and others, which is colloquially called *Squiggol* [STO89, Bir87, Mee86]. In Squiggol, the term "accumulation" has taken on a slightly different meaning. The accumulation of an operator $\oplus$ over a list L, $\oplus\!\not\!\!\twoheadrightarrow_e L$, returns the list containing all intermediate results of a directed reduction, i.e.

$$\begin{aligned} \oplus\!\not\!\!\twoheadrightarrow_e[a_1, a_2, \ldots, a_n] &= [e, e \oplus a_1, \ldots, (\ldots(e \oplus a_1) \oplus a_2 \ldots) \oplus a_n] \\ \oplus\!\not\!\!\twoheadleftarrow_e[a_1, a_2, \ldots, a_n] &= [a_1 \oplus (a_2 \oplus \ldots (a_n \oplus e) \ldots), \ldots, a_n \oplus e, e] \end{aligned}$$

This bears some relation to the accumulation described in [Bir84a], in the sense that the "final result" is built up iteratively. The related optimising transformation is comparable to the relation between reductions on one hand and directed reductions and accumulations on the other hand (cf. [Bir87]). One could also argue that accumulations in the traditional sense are less important in Squiggol because of the use of the (undirected) *reduction* operator / which makes associativity and, often, unit elements, implicit. Informally, the reduction operator / can be characterised by:

$$\oplus/[a_1, a_2, \ldots, a_n] \quad = \quad a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

where the absence of brackets indicates associativity of $\oplus$.

The accumulation strategy as described in this paper is called (one of the applications of) *tupling* in more recent Squiggol terminology. Meertens recently proposed a class of functions called *paramorphisms* [Mee92]. Using a form of tupling, paramorphisms can be transformed into homomorphisms. This construction is also used by Harrison [Har91, fold-axiom R3].

Recent work by Meijer, Fokkinga and Paterson [MFP91] introduces a still wider class of functions, called *hylomorphisms*. Erik Meijer has shown that accumulation can be easily handled in this context [Mei91].

8 Descriptions in CIP-L

In the Munich CIP-project [BBB+85], accumulation is mainly viewed as one of the possibilities for transforming a linear recursive function into tail-recursive form. This can clearly be seen in the textbooks [BW82, Par90].

Bauer and Wössner describe the accumulation with the monoid conditions as a special instance of accumulation with an *associative dual*, i.e., $\oplus$ is of type $P \times Q \to P$, and an operator $\otimes$ (the associative dual of $\oplus$) of type $Q \times Q \to Q$ exists, such that

$$(a \oplus x) \oplus y = a \oplus (x \otimes y).$$

This condition, with the existence of a right identity of $\oplus$, is also sufficient for accumulation (by definition of $f'(x, y) = f(x) \oplus y$). See also [BT90] for a more extensive treatment of associative duals and their properties.

In [Par90], accumulation is described as follows:

Simplification of linear recursion I

funct f = (**m** x)**m**:
  if B(x) **then** H(x) **else** p(K(x),f(K'(x))) **fi**

ASSOC(**m**, p)
NEUTRAL(**m**, p, E)

funct $f = (\mathbf{m}\ x)\mathbf{m}$:
  $f'(\mathrm{E}, x)$ **where**
  funct $f' = (\mathbf{m}\ y, \mathbf{m}\ x)\mathbf{m}$:
    if $\mathrm{B}(x)$ **then** $p(y, \mathrm{H}(x))$ **else** $f'(p(y, \mathrm{K}(x)), \mathrm{K'}(x))$ **fi**

where ASSOC and NEUTRAL denote degenerated algebraic types which state that the function p is associative on type **m** and that E is a neutral element (identity) of the function p on type **m**, respectively. These are the monoid conditions, as expected.

Of course, this transformation rule can also be viewed as the summary of a schematic transformational development, starting from the input scheme of the rule:

Embedding:

define new function f' by

$f'(\mathbf{m}\ y, \mathbf{m}\ x) = p(y, f(x))$

Development:

Focus: function f'

Goal: definition of f' independent of f

Strategy: unfold-fold

Transformations:

step 1: unfold f

$p(y, f($**if** $\mathrm{B}(x)$ **then** $\mathrm{H}(x)$ **else** $p(\mathrm{K}(x), f(\mathrm{K'}(x))$ **fi**$))$

step 2: distribution of function call over conditional

if $\mathrm{B}(x)$ **then** $p(y, \mathrm{H}(x))$ **else** $p(y, p(\mathrm{K}(x), f(\mathrm{K'}(x))))$ **fi**

step 3: associativity of p

if $\mathrm{B}(x)$ **then** $p(y, \mathrm{H}(x))$ **else** $p(\ p(y, \mathrm{K}(x)), f(\mathrm{K'}(x)))$ **fi**

step 4: fold f'

if $\mathrm{B}(x)$ **then** $p(y, \mathrm{H}(x))$ **else** $f'(\ p(y, \mathrm{K}(x)), \mathrm{K'}(x))$ **fi**

The condition NEUTRAL($\mathbf{m}$, p, E) comes in when f is subsequently defined in terms of f'.

Note that the *embedding* (of the domain) is the crucial step in the derivation. Generally, it is well recognised in the CIP-methodology that embedding (or *generalisation*) is a very important method in program transformation.

9 The Edinburgh generalisation strategy

Among the first to recognise the importance of *generalisation* in program transformations was the theorem proving group in Edinburgh [BM75, Aub75]. The generalisation strategy is based on the well-known heuristic that, in order to prove a theorem, one should sometimes prove a more general theorem. In functional programming, this translates to defining more general functions, e.g. functions with extra arguments. The choice between possible generalisations is based on *mismatch information*. Obviously one of the applications of this strategy is *accumulation* (called *generalisation of output terms* in [Aub75]), where the unfold-fold derivation starts with a more general function. In [Aub75], one can also find a derivation of the fast reverse.

10 Accumulation by efficient representation of continuations

In the area of program development, one of the most important issues is reducing the number of *eurekas* or *rabbits* [Bro89, vdW89]. Steps that appear to be inventive need a simple motivation rather than "a little foresight".

In my opinion, one of the best motivations given for accumulation is the one by Wand [Wan80]. In his paper, he describes a simple strategy:

given a function f with argument x, define a new function f' by

$$f'(\gamma, x) \quad = \quad \gamma(f(x)).$$

Thus, $f(x) = f'(\lambda y.y, x)$. γ is a *continuation* function. Then derive a definition of f' independent of f. In many cases, a more efficient representation of the continuation function suggests itself (at this point, there is still a (small?) eureka). This is a generalisation; an (invisible) occurrence of the identity function (a constant) is generalised to an arbitrary function.

Accumulator versions of functions can be derived as follows:
given a function of the form

$$\begin{array}{lcl} f(x) & = & \textbf{if } T(x) \textbf{ then } 1_\oplus \\ & & \textbf{else } f(K(x)) \oplus H(x) \textbf{ fi} \end{array}$$

define

$$f'(\gamma, x) \quad = \quad \gamma(f(x)).$$

then

$$\begin{array}{lll}
f'(\gamma,x) & =_{\{\text{unfold } f\}} & \gamma(\ \textbf{if } T(x) \textbf{ then } 1_\oplus \\
& & \quad \textbf{else } f(K(x)) \oplus H(x) \textbf{ fi}) \\
& =_{\{\text{distributivity}\}} & \textbf{if } T(x) \textbf{ then } \gamma(1_\oplus) \\
& & \textbf{else } \gamma(f(K(x)) \oplus H(x)) \textbf{ fi} \\
& =_{\{\text{abstract}\}} & \textbf{if } T(x) \textbf{ then } \gamma(1_\oplus) \\
& & \textbf{else } (\lambda y.\gamma(y \oplus H(x)))(f(K(x)) \textbf{ fi} \\
& =_{\{\text{fold } f'\}} & \textbf{if } T(x) \textbf{ then } \gamma(1_\oplus) \\
& & \textbf{else } f'(\gamma \circ (\lambda y.y \oplus H(x)), K(x)) \textbf{ fi}.
\end{array}$$

Now it can be shown that every function γ is of the form $\lambda y.y \oplus z$ for some z:

$$\begin{array}{rll}
\lambda y.y & =_{\{\text{trivial}\}} & \lambda y.y \oplus 1_\oplus; \\
\gamma \circ (\lambda y.y \oplus H(x)) & =_{\{\gamma \text{ is } \lambda y.y \oplus z\}} & \lambda y.(y \oplus H(x)) \oplus z \\
& =_{\{\text{associativity } \oplus\}} & \lambda y.y \oplus (H(x) \oplus z)\Box
\end{array}$$

Thus, $\gamma = \lambda y.y \oplus z$ can be represented by just z. Note that $\gamma(1_\oplus)$ simplifies to z. This gives the well-known accumulator version of f:

$$\begin{array}{rcl}
f(x) & = & f''(1_\oplus, x) \\
f''(z,x) & = & \textbf{if } T(x) \textbf{ then } z \\
& & \textbf{else } f''(H(x) \oplus z, K(x)) \textbf{ fi}.
\end{array}$$

It is important to note that there is not much operational difference between the original f and f'. In f' the continuation function represents the actions that have to be taken after the recursion has terminated, while this is left implicit in f. Generally, transformations that change or simplify recursion can be applied by first making explicit part of the recursion mechanism and then proving properties of these representations of recursion (e.g., stacks, cf. the discussion in [Boi92]).

11 Pettorossi's higher order generalisation

Pettorossi and Skowron's *lambda abstraction* or *higher order generalisation* strategy [PS87, Pet87, PS90] takes the ideas from [Wan80] even further. As in the Edinburgh generalisation strategy, *mismatch information* is used to motivate the introduction of functions with functions as extra arguments. Their goal usually is the derivation of tail-recursive functions, as in the CIP methodology.

The application of lambda abstraction to the standard accumulation examples leads to curried versions of the transformed functions.

Often, the lambda abstraction strategy is used in conjunction with the *tupling* strategy [Pet84]. Apart from accumulation, lambda abstraction can also be used to derive programs with continuations, e.g. for deriving complicated programs for traversing datastructures in one pass as in [Bir84b]. Related techniques are described in [Hug82, Joh85, Tak87].

12 Hughes' novel implementation of lists

In [Hug86], the idea is presented that a list x could be represented by

$$rep(x) = \lambda y.x +\!\!\!+ y,$$

with corresponding abstraction function

$$abs(x) = x[\,].$$

It is shown how this allows a straightforward derivation of the "fast reverse" (cf. section 2). Also, concatenation on the abstract level is represented by function composition on the concrete level, since

$$\begin{array}{lll} rep(x +\!\!\!+ y) & =_{\{\text{definition } rep\}} & \lambda z.(x +\!\!\!+ y) +\!\!\!+ z \\ & =_{\{\text{associativity } +\!\!\!+\}} & \lambda z.x +\!\!\!+ (y +\!\!\!+ z) \\ & =_{\{\text{definition } rep\}} & \lambda z.rep(x)(rep(y)z) \\ & =_{\{\eta\text{-conversion}\}} & rep(x) \circ rep(y). \end{array}$$

At a first glance, this seems a revolutionary idea. I will demonstrate that, when one generalises this idea to arbitrary data types with an arbitrary binary operation, it appears that, again, the monoid conditions should hold for this kind of implementations of data types to be useful.

Given a data type D with function $\oplus$ of type $D \times D \to D$ (or $D \to D \to D$), let

$$\begin{array}{lll} rep(d) & = & (d\oplus) \\ abs(x) & = & x.1_{\oplus}. \end{array}$$

This is always a *valid* implementation of D, since

$$abs(rep\ d) = abs(d\oplus) = d \oplus 1_{\oplus} = d.$$

The equation

$$rep(abs\ x) = x$$

only holds for x (a function from $D \to D$) that can be written as $\lambda y.x' \oplus y$:

$$\begin{aligned}
rep(abs\ x) &=_{\{\text{def. } rep\}} && \lambda z.(abs\ x) \oplus z \\
&=_{\{\text{def. } abs\}} && \lambda z.(x\ 1_{\oplus}) \oplus z \\
&=_{\{x \text{ is } \lambda y.x' \oplus y\}} && \lambda z.x' \oplus z \\
&=_{\{\alpha\text{-conversion}\}} && x
\end{aligned}$$

When does this constitute a *useful* implementation of D? The most important aspect is whether $\oplus$ can be efficiently implemented in the new representation:

$$\begin{aligned}
rep(x \oplus y) &=_{\{\text{def. } rep\}} && \lambda z.(x \oplus y) \oplus z \\
&=_{\{\textbf{associativity of } \oplus \textbf{ necessary}\}} && \lambda z.x \oplus (y \oplus z) \\
&=_{\{\text{def. } rep\}} && \lambda z.(rep\ x)(rep\ y)z \\
&=_{\{\eta\text{-conversion}\}} && (rep\ x) \circ (rep\ y).
\end{aligned}$$

For other operators to be efficiently implementable, certain distributive properties should hold.

So, for implementation of objects d of a data type D by a function $d\oplus$, again the monoid properties should hold. An identity of $\oplus$ should exist to allow an abstraction function, and $\oplus$ should be associative for its implementation to be simple. Thus, it is not surprising that this technique led to a derivation of the fast reverse.

13 Some more papers

There are many more papers discussing accumulation, generalisation and related techniques.

Burstall and Feather's early survey paper [BF78] gives the familiar factorial example as an example for generalisation, and more references to papers from the early days of transformational programming.

The classic papers by Strong and others [AS78, Str71, WS73] are mostly concerned with recursion elimination, i.e. with conditions that allow more than the trivial transition from tail recursion to iteration. The associativity conditions are among the most obvious of those.

The paper by Huet and Lang [HL78] uses *recursion removal* as an example of program transformations expressed with second-order patterns. (These second-order patterns are comparable to the transformation rules in the CIP approach). These examples (mainly taken from Darlington's thesis [Dar72]) include accumulation and accumulation with associative dual.

Manna and Waldinger [MW] also mention generalisation as an important technique in program synthesis, and show a derivation of the fast reverse.

Arsac and Kodratoff [AK82] give a method for finding generalisations that lead to tail-recursive functions. Their examples involve more complicated properties than just associativity; the existence of identities of functions is generally assumed (cf. the adjunction of 'fictitious' identity elements in Squiggol [Bir87]). They also briefly consider 'generalisation using second-order ideas', where a functional variable is introduced. By explicitly introducing the stacks that are necessary in the evaluation of non-tail-recursive functions, they derive iterative versions of arbitrary recursive functions.

Gabriel [Gab91] calls the accumulation rule *parentheses movement*[1], and uses it as an example of the DEVA meta-calculus. The most impressive aspect of this paper is the LaTeX-representation of the (really two-dimensional) DEVA fragments, which has been automatically generated by the DEVA system.

Acknowledgement

I thank Alberto Pettorossi for introducing me to many of the papers mentioned above and for interesting discussions on this and other subjects during my stay at IASI in Rome. Helmut Partsch also provided numerous references and useful comments on this paper.

References

[AK82] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, April 1982.

[AS78] M.A. Auslander and M.R. Strong. Systematic recursion removal. *Communications of the ACM*, 21(2):127–133, 1978.

[1]The German version of Bauer and Wössner's book has "Klammerverschiebung".

[Aub75] R. Aubin. Some generalization heuristics in proofs by induction. In *Proc. Int. Symp. on Proving and Improving Programs, Arc-et-Senans, France*, pages 197–208, 1975.

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[BF78] R.M. Burstall and M.S. Feather. Program development by transformation: an overview. In M. Amirchahy and D. Neel, editors, *Les fondements de la programmation. Proc. Toulouse CREST Course on Programming, IRIA-SEFI, Le Chesnay, France*, 1978.

[Bir84a] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir84b] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[Bir87] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, pages 5–42. Springer-Verlag, Berlin, 1987.

[BM75] R.S. Boyer and J.S. Moore. Proving theorems about LISP functions. *Communications of the ACM*, 22(1):129–144, 1975.

[Boi92] E.A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18:139–179, 1992.

[Bro89] M. Broy, editor. *Constructive Methods in Computing Science. NATO ASI Series Vol. F55*, Berlin, 1989. Springer-Verlag.

[BT90] E.A. Boiten and D. Tuijnman. Properties and application of associative duals. Unpublished note, 1990.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, Berlin, 1982.

[Coo66] D.C. Cooper. The equivalence of certain computations. *Computer Journal*, 9:45–52, 1966.

[Dar72] J. Darlington. *A semantic approach to automatic program improvement*. PhD thesis, Dept. of Machine Intelligence, University of Edinburgh, 1972. Summarized in [DB76].

[DB76] J. Darlington and R.M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[Fea87] M.S. Feather. 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*, pages 165–196, Amsterdam, 1987. North-Holland Publishing Company.

[Gab91] R. Gabriel. Program transformation expressed in DEVA meta-calculus. In Möller [Möl91], pages 267–285.

[Gil76] Arthur Gill. *Applied Algebra for the Computer Sciences*. Prentice-Hall, 1976.

[Har91] P.G. Harrison. Towards the synthesis of static parallel algorithms: a categorical approach. In Möller [Möl91], pages 49–69.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[Hug82] John Hughes. Super-combinators: A new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, Pittsburgh, August 1982.

[Hug86] R.J.M. Hughes. A novel implementation of lists and its application to the function 'reverse'. *Information Processing Letters*, 22:141–144, 1986.

[Joh85] T. Johnsson. Lambda-lifting: transforming programs to recursive equations. In J.P. Jouannaud, editor, *Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Berlin, 1985. Springer-Verlag.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[Mee86] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334, 1986.

[Mee92] L.G.L.T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[Mei91] E. Meijer. Abreasting accumulating arguments. Unpublished note, May 1991.

[MFP91] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Möl91] B. Möller, editor. *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, Amsterdam, 1991. North-Holland Publishing Company.

[MW] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis.

[Par90] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[Pet84] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on LISP and Functional Programming*, pages 273–281, 1984.

[Pet87] A. Pettorossi. Program development using lambda abstraction. In *Proc. 7th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science, Pune, India, 1987*, volume 287 of *Lecture Notes in Computer Science*, pages 401–434, Berlin, 1987. Springer-Verlag.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[PS87] A. Pettorossi and A. Skowron. Higher order generalization in program derivation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87 — Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 250 of *Lecture Notes in Computer Science*, pages 182–196, Pisa, March 1987. Springer-Verlag.

[PS90] A. Pettorossi and A. Skowron. The lambda abstraction strategy for program derivation. To be submitted for publication, 1990.

[STO89] STOP. *STOP International Summer School on Constructive Algorithmics, Ameland*, September 1989. Lecture notes.

[Str71] H.R. Strong, Jr. Translating recursion equations into flowcharts. *Journal of Computer and System Sciences*, 5:254–285, 1971.

[Tak87] Masato Takeichi. Partial parametrization eliminates multiple traversals of data. *Acta Informatica*, 24:57–77, 1987.

[vdW89] J. van der Woude. Rabbitcount := Rabbitcount – 1. In J.L.A. Van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 409–420, Berlin, 1989. Springer-Verlag.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[WS73] S.A. Walker and H.R. Strong. Characterizations of flowchartable recursions. *Journal of Computer and System Sciences*, 7:404–447, 1973.