Higher-order Transformations and Type Simulations

M. C. HENSON
Department of Computer Science, University of Essex, Wivenhoe Park, Colchester, Essex, CO4 3SQ

We examine higher-order program transformations in the context of a typed, pattern-directed, higher-order, lazy functional language. We introduce the notions of higher-order accumulation and type simulation as transformational techniques. Our main illustration of these techniques concerns the transformation of a description of a small programming language into a code generator. In this example we show how our techniques tackle important questions regarding the circumstances under which data, in a functional program, can be destructively updated with impunity.

Received April 1987, revised February 1988

1. INTRODUCTION

In this paper we illustrate two techniques for program development by transformation. These are 'higher-order accumulation' and 'type simulation'. We introduce these ideas in the next section by examining a fairly small and elementary example concerned with tree processing. In the remainder of the paper we tackle a more substantial example: we derive a compiler (more exactly a code generator) from the functional semantics of a toy programming language. We shall pay particular attention in this example to the interesting and important topic of identifying data, in a functional program, which can be destructively updated with impunity.

Transformational programming is a methodology for program development in which specifications or inefficient programs are massaged into efficient final forms by a number of correctness-preserving program transformations. Perhaps the most influential work in this area was pioneered by Burstall and Darlington.[1-3] A central concern is that all the permitted transformational steps preserve meaning. In this way we can be sure that the final program and the original specification are equivalent. Under certain conditions Burstall and Darlington's transformations do preserve correctness. In fact it has been realised for some time that such transformations are essentially inductive proofs of program equivalence in another guise. The advantage of the transformational approach is evident: with transformation the proof and program are derived simultaneously, whilst with an inductive proof of equivalence we already need the target program to have been (somehow) constructed.

The heart of program transformation in the Burstall–Darlington paradigm is the notion of a eureka definition. Such definitions are crucial to successful program improvement but are by no means easy to construct. To be sure, a number of techniques or styles of eureka definition have been discovered, often corresponding to well-known programming techniques, but in general eureka definitions isolate the creative input which is necessary for successful programming.

Higher-order accumulation is one such style of eureka definition. It has its origins in the continuation semantics of Strachey and the transformational work of Wand[12,14] It is, in a sense, a brute-force strategy, since it can be applied uniformly to any functional program and the result of applying it is rarely a particularly impressive program. Type simulation, however, is in general a more subtle technique, which requires greater creativity and which, with care, produces improved programs. These techniques are complementary, because it is the type of the extra datum introduced by higher-order accumulation which is simulated in a second transformation in the examples we will consider here. In earlier work[6] we have shown how these techniques subsume a still wider class of otherwise distinct program development strategies including (first-order) accumulation, generalisation, recursion elimination and some aspects of exception handling and backtracking.

The programming notation we employ is a language of typed, higher-order, pattern-directed recursion equations with a lazy semantics. It resembles the language Miranda.[13] Our notation for types differs somewhat from that of Miranda but is easily grasped. The reason is that we need to state many properties of polymorphic functions and relations. This is best done within a type theory. We will not make this theory explicit, relying instead on a second-order logical notation. We write $A \to B$ for the type of functions from type A to type B. $A \times B$ is the ordinary cartesian product type and $A + B$ is ordinary (disjoint) union. Finally we introduce the notion of a variable over types (which we write with Greek letters) and then $\Pi\alpha\,.\,t$ is the polymorphic type t over α. Objects of this type satisfy the axiom: $t \in \Pi\alpha\,.\,\beta \Leftrightarrow (\forall\tau)\,(t \in \beta[\alpha \leftarrow \tau])$. For example, suppose we have a function, **reverse** which, given a list, reverses it. Its type is: $\Pi\alpha\,.\,\mathrm{List}\,(\alpha) \to \mathrm{List}\,(\alpha)$, that is, $\mathrm{List}\,(t) \to \mathrm{List}\,(t)$ for any type t. This type, because it can reverse lists of any element type. We need two type constructors called List and Sexp which given a type t yield lists whose elements are in t and binary trees whose leaves are in t, respectively. These are given as follows:

$$\mathrm{List}\,(\alpha) = \{[\,]\} + \alpha \times \mathrm{List}\,(\alpha)$$
$$\mathrm{Sexp}\,(\alpha) = \alpha + \mathrm{Sexp}\,(\alpha) \times \mathrm{Sexp}\,(\alpha)$$

Data constructors are the names given to form tuples of a cartesian product type. For example, we will take it that the product operation in the definition of List uses an infix colon and the product in Sexp uses an infix double colon. Thus, 3:4:[ ] is an element of $\mathrm{List}\,(N)$ (where we take it that ':' associates to the right and, in future, that 3:4:[ ] and other lists will be sugared in an obvious way to [3 4]). Moreover, 3::4 is an element of $\mathrm{Sexp}\,(N)$.

We will use the ordinary set membership relation, $\in$, to indicate the relation 'has the type' in the rest of the paper.

2. HIGHER-ORDER ACCUMULATION AND TYPE SIMULATION

In this section we will introduce the techniques of higher-order accumulation and type simulation by applying them to a simple program scheme for computing with trees.

Definition 2.1 – Higher-order accumulation

Let $\mathbf{f} \in (\Pi\alpha)(\Pi\beta)(\alpha \to \beta)$. We associate with this a function: $\mathbf{f}' \in (\Pi\alpha)(\Pi\beta)((\alpha \to \beta) \to \beta)$ by means of the following definition:

$\mathbf{f}' x k = (k \circ \mathbf{f}) x$

where $\circ$ is functional composition. ■

Corollary 2.2

Let $k, k' \in (\Pi\alpha)(\alpha \to \alpha)$, let $\mathbf{f} \in (\Pi\alpha)(\Pi\beta)(\alpha \to \beta)$ and $\mathbf{f}' \in (\Pi\alpha)(\Pi\beta)(\alpha \to (\beta \to \beta) \to \beta)$, then, for all x:

$k(\mathbf{f}' x k') = (\mathbf{f}' x (k \circ k'))$. ■

The definition of $\mathbf{f}'$ in terms of $\mathbf{f}$ is taken to be a eureka definition and can be used to undertake a transformation.

We begin with a classic structural recursive scheme over trees or, for the sake of simplicity, binary trees. Recall that $\mathrm{Sexp}(\alpha)$ is the type of s-expressions over α. We define a program:

$\mathbf{j} \in (\Pi\alpha)(\Pi\beta)((\alpha \to \beta) \times (\beta \times \beta \to \beta) \times \mathrm{Sexp}(\alpha) \to \beta)$

$\mathbf{j} d h (s_1 :: s_2) = h(\mathbf{j} g h s_1)(\mathbf{j} g h s_2)$

$\mathbf{j} d h a = d a$

where h is associative and has a unit, 1_h.

In fact, it is more convenient to work with the simpler program scheme, in which we take d and h to be program variables of the appropriate types. For convenience, we will write the program variable, h, as an infix, $\oplus$.

$\mathbf{f} \in (\Pi\alpha)(\Pi\beta)(\mathrm{Sexp}(\alpha) \to \beta)$

$\mathbf{f}(s_1 :: s_2) = (\mathbf{f} s_1) \oplus (\mathbf{f} s_2)$

$\mathbf{f} a = d a$

The task of transforming this program scheme into an iterative system of recursion equations was originally undertaken in the seminal paper by Burstall and Darlington.[1]

We attack this with higher-order accumulation, obtaining a new function, $\mathbf{f}' \in (\Pi\alpha)(\Pi\beta)(\mathrm{Sexp}(\alpha) \to (\beta \to \beta) \to \beta)$ by setting:

$\mathbf{f}' s k = (k \circ \mathbf{f}) s.$

We show just one step in the transformation of this program into a form independent of $\mathbf{f}$; that for a compound expression, $s_1 :: s_2$. This proceeds as follows.

$\mathbf{f}'(s_1 :: s_2) k =$

$(k \circ \mathbf{f})(s_1 :: s_2) =$

$k((\mathbf{f} s_1) \oplus (\mathbf{f} s_2)) =$

$k((\lambda z . z \oplus (\mathbf{f} s_2))(\mathbf{f} s_1)) =$

$(k \circ (\lambda z . z \oplus (\mathbf{f} s_2)) \circ \mathbf{f}) s_1 =$ (fold)

$\mathbf{f}' s_1 (k \circ (\lambda z . z \oplus (\mathbf{f} s_2))) =$

$\mathbf{f}' s_1 (k \circ (\lambda z . (((\lambda z' . z \oplus z') \circ \mathbf{f}) s_2))) =$ (fold)

$\mathbf{f}' s_1 (k \circ (\lambda z . \mathbf{f}' s_2 \lambda z' . z \oplus z')) =$ (corollary 2.2)

$\mathbf{f}' s_1 (\lambda z . \mathbf{f}' s_2 (\lambda z' . k(z \oplus z')))$

The steps labelled 'fold' are justified because s_1 and s_2 are proper subexpressions of $s_1 :: s_2$. The complete transformation yields the following system of equations:

$\mathbf{f} s = \mathbf{f}' s \mathbf{id}$

$\mathbf{f}'(s_1 :: s_2) = \mathbf{f}' s_1 (\lambda z . \mathbf{f}' s_2 (\lambda z' . k(z \oplus z')))$

$\mathbf{f}' a k = k(d a)$

As we indicated in the introduction, the technique is uniform and does not involve any creativity for its application. We now turn to the next stage, which involves the simulation of the argument k, which now occurs in our transformed program $\mathbf{f}'$. We begin by observing that the class of functions which can occur as accumulators in $\mathbf{f}'$ may be given by an inductive definition.

Definition 2.3

$(\forall\alpha)(\forall\beta)(\forall k \in \beta \to \beta)\ (\mathrm{acc}(k) \Leftrightarrow_{\mathrm{def}} k = \mathbf{id} \vee$

$(\exists s \in \mathrm{Sexp}(\alpha))(\exists k' \in \beta \to \beta)\ (\mathrm{acc}(k') \wedge k =$

$\lambda z . \mathbf{f}' s (\lambda z' . k'(z \oplus z'))))$. ■

It is now possible to investigate the accumulators of $\mathbf{f}'$. We look for some simple property which will form the basis for a type simulation.

Proposition 2.4

$(\forall\alpha)(\forall k \in \alpha \to \alpha)(\mathrm{acc}(k) \Rightarrow (\exists z \in \alpha)$

$(\forall z' \in \alpha)((k z') = z' \oplus z))$

Proof

Since accumulators are given by an inductive definition we may establish this by induction on their structure. We also make use of the properties of the operator $\oplus$. ■

This result suggests a relational connection between certain lists and accumulators which spells out when a list can be said to simulate an accumulator. We are thus led to

Definition 2.5

$(\forall\alpha)(\forall k \in \alpha \to \alpha)(\forall z \in \alpha)(z \approx k \Leftrightarrow_{\mathrm{def}}$

$(\forall z' \in \alpha)((k z') = z' \oplus z)$. ■

Corollary 2.6

$z \approx \lambda z' . z' \oplus z$ whence $1_{\oplus} \approx \mathbf{id}$. ■

We now stipulate a constraint on a proposed function $\mathbf{g}$. This is the *specification* of $\mathbf{g}$ and note that it is *not* a program – it is a requirement which must be satisfied. In spite of this we still derive $\mathbf{g}$ from the specification by transformation:

$\mathbf{g} \in (\Pi\alpha)(\Pi\beta)(\mathrm{Sexp}(\alpha) \to \beta \to \beta)$

$(\forall\alpha)(\forall\beta)(\forall k \in \beta \to \beta)(\forall z \in \beta)$

$(z \approx k \Rightarrow ((\forall s \in \mathrm{Sexp}(\alpha))(\mathbf{g} s z = \mathbf{f}' s k)))$

Surprisingly, the lack of a functional connection between $\mathbf{g}$ and $\mathbf{f}'$ does not present a problem. A technical remark:

the transformational step called folding, under this form of eureka constraint, is only reasonable when the argument of the expression being folded is lower in some well-ordering than that of the equation being transformed. We proceed as follows.

Suppose that $z \approx k$:

$\mathbf{g}(s_1::s_2)\,z =$

$\mathbf{f}'(s_1::s_2)\,k =$

$\mathbf{f}'s_1(\lambda z'.\mathbf{f}'s_2(\lambda z''.k(z' \oplus z''))) =$ (assumption)

$\mathbf{f}'s_1(\lambda z'.\mathbf{f}'s_2(\lambda z''.z' \oplus z'' \oplus z)) =$ (Corollary 2.2)

$\mathbf{f}'s_1(\lambda z'.z' \oplus (\mathbf{f}'s_2(\lambda z''.z'' \oplus z))) =$ (fold)

$\mathbf{f}'s_1(\lambda z'.z' \oplus (\mathbf{g}s_2\,z)) =$ (fold)

$\mathbf{g}\,s_1(\mathbf{g}\,s_2\,z)$

Suppose that $z \approx k$:

$\mathbf{g}\,a\,z =$

$\mathbf{f}'a\,k =$

$k(d\,a) =$ (assumption)

$(d\,a) \oplus z$

Since $1_\oplus \approx id$:

$\mathbf{f}\,s =$

$\mathbf{f}'s\,\mathbf{id} =$

$\mathbf{g}\,s\,1_\oplus$

In summary we obtain the following system of equations:

$\mathbf{f}\,s = \mathbf{g}\,s\,1_\oplus$

$\mathbf{g}(s_1::s_2)\,z = \mathbf{g}\,s_1(\mathbf{g}\,s_2\,z)$

$\mathbf{g}\,a\,z = (d\,a) \oplus z$

If we take '$\oplus$' to be **append** (and so $1_\oplus = [\,]$), d, to be **list** (**list** $s = [s]$) then **f** computes the 'fringelist' of its argument. Its space complexity (due to data structure creation) lies between quadratic and linear (with $n.\log n$ being the average case). The new version of **f**, in terms of **g**, is linear, independent of argument shape; indeed it uses half the space required by the original program in the best case.

In order to demonstrate the relationship between the simulation and the final program we now show that there is another way to characterise continuations of **f'** and that this leads to another version of the program **f**.

Definition 2.7

$\mathrm{Rep} \in (\Pi\alpha)(\Pi\beta)(\mathrm{List}(\mathrm{Sexp}(\alpha)) \to \beta \to \beta)$

$\mathrm{Rep}[\,] = \mathbf{id}$

$\mathrm{Rep}(s{:}l) = \lambda z.\mathbf{f}'s(\mathrm{Rep}\,l) \circ (\lambda z'.z \oplus z')$. ■

Proposition 2.8

Accumulators, k, of **f'** have the following property:

$(\forall\alpha)(\forall k \in \alpha \to \alpha)$

$(\mathrm{acc}(k) \Rightarrow (\exists l \in \mathrm{List}(\mathrm{Sexp}(\alpha)))((\mathrm{Rep}\,l) = k))$

Proof

As before we make use of the inductive structure of the accumulators. ■

This suggests another relation.

Definition 2.9

$\approx \subseteq (\Pi\alpha)(\Pi\beta)(\mathrm{List}(\mathrm{Sexp}(\alpha)) \times (\beta \to \beta))$

$(\forall\alpha)(\forall\beta)(\forall k \in \beta \to \beta)(\forall l \in \mathrm{List}(\mathrm{Sexp}(\alpha))$

$(l \approx k \Leftrightarrow_{\mathrm{def}} ((\mathrm{Rep}\,l) = k))$ ■

Corollary 2.10

$[\,] \approx \mathbf{id}$ ■

We can introduce a new constraint on a proposed function, **ff**:

$\mathbf{ff} \in (\Pi\alpha)(\Pi\beta)(\mathrm{List}(\mathrm{Sexp}(\alpha)) \to \beta)$

$(\forall\alpha)(\forall\beta)(\forall k \in \beta \to \beta)(\forall l \in \mathrm{List}(\mathrm{Sexp}(\alpha))$

$(l \approx k \Rightarrow ((\mathbf{ff}\,l) = (k1_\oplus)))$

Note that if we define **ff** as follows, the constraint is always satisfied:

$\mathbf{ff}[\,] = 1_\oplus$

since $[\,] \approx \mathbf{id}$ and $(\mathbf{id}\ 1_\oplus) = 1_\oplus$.

$\mathbf{ff}(s{:}l) = \mathbf{f}'s(\mathrm{Rep}\,l)$

since, if $(s{:}l) \approx k$ then $k = (\mathrm{Rep}(s{:}l))$ and so

$k = (\lambda z.\mathbf{f}'s(\mathrm{Rep}\,l) \circ (\lambda z'.z \oplus z'))$.

Now

$(k\,1_\oplus) = \mathbf{f}'s(\mathrm{Rep}\,l) \circ \lambda z.1_\oplus \oplus z = \mathbf{f}'s(\mathrm{Rep}\,l) \circ \mathbf{id}$.

This now establishes a functional formulation of the relational constraint and we can transform with respect to it. The transformation step when the argument is a leaf is most troublesome. We require the following lemmata.

Lemma 2.11

$(\forall\alpha)(\forall\beta)(\forall x \in \mathrm{List}(\mathrm{Sexp}(\alpha))(\forall y, z \in \beta)$

$(\mathrm{Rep}\,x(y \oplus z)) = y \oplus (\mathrm{Rep}\,x\,z)$ ■

This is a version of Corollary 2.2 at the level of representation functions.

Lemma 2.12

$(\forall\alpha)(\forall z, z' \in \alpha)((\mathrm{Rep}\,z\,z') = z' \oplus (\mathbf{ff}\,z))$

Proof

This follows from Lemma 2.12 and Corollary 2.2. ■

After the transformation we obtain the new version of **f**, which is:

$\mathbf{f}\,s = \mathbf{ff}[s]$

$\mathbf{ff}[\,] = 1_\oplus$

$\mathbf{ff}((s_1::s_2){:}l) = \mathbf{ff}(s_1{:}(s_2{:}l))$

$\mathbf{ff}(a{:}l) = (d\,a) \oplus (\mathbf{ff}\,l)$

The programming technique which is here subsumed by the continuation transform is usually called general-

isation. Under this advice one takes a program of type $A \to B$ and introduces a new one of type $C \to B$ where there is an embedding $i \in A \to C$. In our example $A =$ Sexp (α) and $C =$ List (Sexp (α)) so $i \in (\Pi\alpha)$ (Sexp $(\alpha) \to$ List (Sexp (α))) is just: $is = [s]$.

This device was used by Burstall and Darlington[1] to avoid overcomputation in the classic 'samefringe' problem.[4] The program **ff** can if required be transformed into an iteration by adding an accumulator for the prefix: '$(ga) \oplus$' in the final equation.

3. STACK EXTRACTION BY HIGHER-ORDER ACCUMULATION AND TYPE SIMULATION

A criticism, often head, of functional languages is their inability to make destructive updates to data structures, leading to poor space utilisation. The truth is that a functional program may well make a destructive update (if it is safe to do so) but there is, however, no way of articulating this within the language. An example of safe destructive updating is the optimisation of tail recursion to iteration. There has also been research aimed at establishing criteria by which certain data structures in a functional program might be destructively updated.[7,9] Schmidt defines a datum as 'single-threaded' if it is possible to replace it by 'access rights to a single global variable while preserving operational properties'.

Often, in a functional program, one deals with an argument which is definitely not single-threaded. Perhaps, however, it is possible to transform the program into one in which the argument is single-threaded. Our example demonstrates how the combination of higher-order accumulation and type simulation can factor out singly threaded data objects from a multiple-threaded context, and concerns the notion of a symbol table or environment. Such entities are classic instances of multiple-threaded data objects which can be recast in a single-threaded fashion by generalising them to stacks. We investigate this by considering a very small imperative programming language with just sufficient content to cause us the problem that we wish to tackle.

3.1 Syntax

$c \in$ CMD, $d \in$ DEC, $e \in$ EXP, $i \in$ Var, $n \in$ NUM

$c ::=$ '$i := e$' | '$c;c$' | 'begin $d;c$ end'

$e ::=$ 'n' | 'i' | '$e+e$'

$d ::=$ 'var i' | '$d;d$'

Commands are assignments, sequences or blocks (block structuring being the most important for the purposes of this example). Expressions are numerals, identifiers and sums. Declarations are individuals or sequences.

It is not difficult to simulate parse trees of this small language as elements of types in our notation, with suitable data constructors for various components. We will not do this explicitly, but from now on we will treat the domains CMD, EXP, DEC, NUM and VAR as these types.

We will define a function (in fact a recursive family of functions) which expounds the evaluation of these programs. We begin by defining the types required to model evaluation.

Let $S =$ List (LOC $\times$ (N + {'Unactivated'})) be the type from which the store is obtained. That is, an element of type S represents the store with respect to which programs are executed. Let ENV = List (VAR $\times$ LOC) be the type of environments from which the bindings of variables to locations can be obtained. It is elements of this type which will concern us most. For both ENV and S we will write the obvious 'lookup' operations as though they are function applications. This keeps the presentation simple and clear.

We now introduce three evaluation functions for the three syntactic classes. The techniques employed for this are just those well-known principles from denotational semantics.[10,11]

cmd $\in$ CMD $\to$ ENV $\to S \to S$

exp $\in$ EXP $\to$ ENV $\to S \to N$

dec $\in$ DEC $\to$ ENV $\to$ S $\to$ ENV $\times S$

cmd $(i := e)\, p\, s =$ **update** $s\,(p\, i)$ (**exp** $e\, p\, s$)

cmd $(c;c')\, p\, s =$ **cmd** $c'\, p$ (**cmd** $c\, p\, s$)
cmd (begin $d;c$ end) $p\, s =$ **let** $p' : s' =$ (**dec** $d\, p\, s$) **in** (**cmd** $c\, p'\, s'$)

We will not bother with the details of **update** except to say that it is a polymorphic function (we will use it again in another type context). The **let** expression is just syntactic sugar in the following sense:

let $x = e$ **in** $e' = (\lambda x.e')\, e$

An important corollary of this definition which we will have occasion to use is:

Corollary 3.1

Assuming that x is not free in e'';
let $x = e$ **in** (**let** $y = e'$ in e'') = **let** $y =$ (**let** $x = e$ in e') in e'' ■

Note that the second and third equations above prevent the argument p from being a single thread; p appears for both subsequents and indeed may, by the third equation, be altered by either or both.

exp $n\, p\, s = n$

exp $i\, p\, s = s\,(p\, i)$

exp $(e+e')\, p\, s =$ **plus** (**exp** $e\, p\, s$) (**exp** $e'\, p\, s$)

dec (var i) p s = (**update** $p\, i$ (**new** s)) : (**activate** (**new** s) s)

dec $(d;d')\, p\, s =$ **let** $p' : s' =$ (**dec** $d\, p\, s$) **in** (**dec** $d'\, p'\, s'$)

We will not go into the details of the functions **new** (which obtains the first unactivated location in its store argument) and **activate**, which marks a location as activated (= not unactivated).

It is possible by careful presentation to see that the last argument of these three functions is in fact safe for destructive updates. This is not immediate (see the last equation for **exp** for example) but we go into no further detail. The reader should consult Ref. 9 for syntactic criteria for establishing single threading. It is comforting that this *is* the case, because one expects the store of an imperative language to be destructively updated!

Instead we tackle the harder problem of transforming this collection of functions into a form in which the

environment appears as a singly threaded entity too. Our first stage is to apply higher-order accumulation simultaneously to the first functions.

We introduce three new functions, **Cmd**, **Exp** and **Dec** as follows:

$\mathbf{Cmd}\, c\, p\, q\, s = (q \circ (\mathbf{cmd}\, c\, p))\, s$

$\mathbf{Exp}\, e\, p\, k\, s = (k \circ (\mathbf{exp}\, e\, p))\, s$

$\mathbf{Dec}\, d\, p\, x\, s = (x \circ (\mathbf{dec}\, d\, p))\, s$

These eureka definitions induce the types of the new functions and, in particular, the three kinds of continuations:

$\mathbf{Cmd} \in \mathrm{CMD} \to \mathrm{ENV} \to \mathrm{CCON} \to S \to S$

$\mathbf{Exp} \in \mathrm{EXP} \to \mathrm{ENV} \to \mathrm{ECON} \to S \to S$

$\mathbf{Dec} \in \mathrm{DEC} \to \mathrm{ENV} \to \mathrm{DCON} \to S \to S$

where

$\mathrm{CCON} = S \to S$, $\mathrm{Econ} = N \to S \to S$ and

$\mathrm{DCON} = \mathrm{ENV} \times S \to S$.

In fact we will curry DCON, obtaining $\mathrm{DCON} = \mathrm{ENV} \to S \to S$, and this requires us to alter the eureka definition to:

$\mathbf{Dec}\, d\, p\, x\, s = \mathbf{let}\, p' : s' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, x\, p'\, s'$

Armed with these we can transform, utilising the natural substructure ordering on the parse trees as the well-ordering necessary for the fold steps. We give two of the interesting transformation steps.

$\mathbf{Cmd}$ (begin d; c end) $p\, q\, s =$

$(q \circ (\mathbf{cmd}$ (begin d; c end) $p))\, s =$

$(q \circ (\lambda s.\, \mathbf{let}\, p' : s' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, (\mathbf{cmd}\, c\, p'\, s')))\, s =$

$(q \circ (\lambda s.\, (\mathbf{Dec}\, d\, p\, (\mathbf{cmd}\, c)\, s)))\, s =$

$\mathbf{Dec}\, d\, p\, (q \circ (\mathbf{cmd}\, c))\, s =$

$\mathbf{Dec}\, d\, p\, (\lambda p'.\, \mathbf{Cmd}\, c\, p'\, q)\, s$

$\mathbf{Dec}\, (d; d')\, p\, x\, s =$

$\mathbf{let}\, p' : s' = (\mathbf{dec}\, (d; d')\, p\, s)\, \mathbf{in}\, x\, p'\, s' =$

$\mathbf{let}\, p' : s' = (\mathbf{let}\, p'' : s'' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, (\mathbf{dec}\, d'\, p''\, s''))\, \mathbf{in}\, x\, p'\, s'$

$\mathbf{let}\, p'' : s'' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, (\mathbf{let}\, p' : s' = (\mathbf{dec}\, d'\, p''\, s'')\, \mathbf{in}\, x\, p'\, s')$

$\mathbf{let}\, p'' : s'' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, (\mathbf{Dec}\, d'\, p''\, s''\, x) =$

$\mathbf{let}\, p'' : s'' = (\mathbf{dec}\, d\, p\, s)\, \mathbf{in}\, (\lambda p'.\, \mathbf{Dec}\, d'\, p'\, x)\, p''\, s'' =$

$\mathbf{Dec}\, d\, p\, (\lambda p'.\, \mathbf{Dec}\, d'\, p'\, x)\, s$

Note the use of Corollary 3.1 in the second of these steps. After the transformation we have the following new version of the three functions:

$\mathbf{Cmd}\, (i := e)\, p\, q\, s = \mathbf{Exp}\, e\, p\, (\lambda n.\, \lambda s.\, (q(\mathbf{update}\, s\, (p\, i)\, n)))\, s$

$\mathbf{Cmd}\, (c; c')\, p\, q\, s = \mathbf{Cmd}\, c\, p\, (\mathbf{Cmd}\, c'\, p\, q)\, s$

$\mathbf{Cmd}\, (\mathrm{begin}\, d; c\, \mathrm{end})\, p\, q\, s = \mathbf{Dec}\, d\, p\, (\lambda p'.\, \mathbf{Cmd}\, c\, p\, q) = s$

$\mathbf{Exp}\, n\, p\, k\, s = k\, n\, s$

$\mathbf{Exp}\, i\, p\, k\, s = k(s(p\, i))\, s$

$\mathbf{Exp}\, (e + e')\, p\, k\, s = \mathbf{Exp}\, e\, p\, (\lambda n.\, \mathbf{Exp}\, e'\, p\, (\lambda n'.\, k(\mathbf{plus}\, n\, n')))$

$\mathbf{Dec}\, (\mathrm{var}\, i)\, p\, x\, s = x(\mathbf{update}\, p\, i\, (\mathbf{new}\, s))\, (\mathbf{activate}\, (\mathbf{new}\, s)\, s)$

$\mathbf{Dec}\, (d; d')\, p\, x\, s = \mathbf{Dec}\, d\, p\, (\lambda p'.\, \mathbf{Dec}\, d'\, p'\, x)\, s$

This new model of the evaluation is well known to semanticists as the continuation semantics for the language. It is particularly useful because we can formulate an algebraic presentation which highlights certain constraints on the argument we are looking to factor out.

We now define the family of functions and infix binary connectives. Note that we now overload the symbol '$\to$', which is one of these connectives. The sense of the symbol should be clear from the contexts in which it appears.

Let us set $\mathrm{CM} = \mathrm{ENV} \to \mathrm{CCON} \to \mathrm{S} \to \mathrm{S}$, $\mathrm{EM} = \mathrm{ENV} \to \mathrm{ECON} \to \mathrm{S} \to \mathrm{S}$ and $\mathrm{DM} = \mathrm{ENV} \to \mathrm{DCON} \to \mathrm{S} \to \mathrm{S}$ and let m range over CM, r over EM and v over DM.

$m \to m' = \lambda pqs.\, mp(m'pq)\, s$

$r \Rightarrow_n r' = \lambda pkz_1 \ldots z_n s.\, rp(r'pk)\, z_1 \ldots z_n\, s$

$v \to > v' = \lambda pxs.\, vp(\lambda p'.\, v'p'x)\, s$

$v \to \longrightarrow m = \lambda pqs.\, vp(\lambda p'.\, mp'q)\, s$

$r \Rightarrow \Longrightarrow f = \lambda pqs.\, rp(\lambda n.\, fnpqs)$

$\mathbf{ADD}\, k\, n\, n'\, s = k(\mathbf{plus}\, n\, n')\, s$

$\mathbf{PUSH}\, n\, k\, s = k\, n\, s$

$\mathbf{LOOKUP}\, i\, p\, k\, s = k(s(p\, i))\, s$

$\mathbf{DECL}\, i\, p\, x\, s = x(\mathbf{update}\, p\, i(\mathbf{new}\, s))\, (\mathbf{activate}\, (\mathbf{new}\, s)\, s)$

$\mathbf{ASSIGN}\, i\, n\, p\, q\, s = q(\mathbf{update}\, s(p\, i)\, n)$

These allow us to rewrite the three functions above as follows:

$\mathbf{Cmd}\, (i := e) = (\mathbf{Exp}\, e) \Rightarrow \Longrightarrow (\mathbf{ASSIGN}\, i)$

$\mathbf{Cmd}\, (c; c') = (\mathbf{Cmd}\, c) \to \to (\mathbf{Cmd}\, c')$

$\mathbf{Cmd}\, (\mathrm{begin}\, d; c\, \mathrm{end}) = (\mathbf{Dec}\, d) \to \longrightarrow (\mathbf{Cmd}\, c)$

$\mathbf{Exp}\, n = \mathbf{PUSH}\, n$

$\mathbf{Exp}\, i = \mathbf{LOOKUP}\, i$

$\mathbf{Exp}\, (e + e') = (\mathbf{Exp}\, e) \Rightarrow_0 (\mathbf{Exp}\, e') \Rightarrow_1 \mathbf{ADD}$

$\mathbf{Dec}\, (\mathrm{var}\, i) = \mathbf{DECL}\, i$

$\mathbf{Dec}\, (d; d') = (\mathbf{Dec}\, d) \to > (\mathbf{Dec}\, d')$

This algebraic technique has been investigated by Raoult and Sethi[8] and by Wand.[15] Wand pioneered the investigation of the properties of these kinds of semantic algebra.

One property which is easy to establish is the following:

Proposition 3.2

$(\forall m, m', m'' \in \mathrm{CM});\ (m \to m') \to m'' = m \to (m' \to m'')$ ■

What is the significance of such associativity laws? Wand uses them to obtain code generators by rotating semantic trees into instruction sequences by sufficient applications of left-to-right associativity. We will see later that his technique can be made to work on our example; but it will not at present because there are

insufficient associativity laws. More particularly we are interested in the algebra of these higher-order connectives because they highlight the ways in which contextual information, which we can take to mean any data objects implicitly processed by the connectives, can be manipulated. To be more specific we turn to an interesting example.

Consider the program fragment: begin d; c end; c'. In terms of the algebra of connectives this is: $(d \rightarrow \longrightarrow c) \rightarrow c'$. Is there an associativity law for such expressions? The 'obvious' choice: $d \rightarrow \longrightarrow (c \rightarrow c')$ is not equivalent to the earlier expression. Intuitively this happens because in the latter case the subcommand c' obtains an environment enriched by the declarations in d whilst the former one avoids this. The failure of this property is necessary; indeed it is the way that the algebra of connectives enforces scope control.

Indeed there seems to be no way to define a new connective which would solve the problem: Suppose that '$\oplus$' is a connective such that: $(v \rightarrow \longrightarrow c) \rightarrow c = v \oplus (c \rightarrow c')$. Clearly, $\oplus$ will have to communicate two environments to the sub-expression $(c \rightarrow c')$ in order that the two sub-expressions of *this* are evaluated properly. A little further thought should reveal that in the case of iterated block structure (a common situation) the connective $\oplus$ might have to communicate a host of environments to its right sub-expression.

This counter-argument contains the seeds of a solution. If the environment was represented as a list of 'small environments' (one for each nested block) the problem could be overcome by means of operations on these new environments (called **NEW** and **OLD**) which add or subtract layers.

We are led to consider the introduction of a new data type of ensembles, defined as follows: ENS = List (ENV) and then to obtain a new family of functions in terms of this. Since we are about to transform the definition we might as well take the opportunity to introduce a partial result stack too, given by the type: STK = List(N) in order to avoid the family of connectives, $\Rightarrow_n$. In other words, rather than deal with arbitrary numbers of partial result arguments, we carry around a single stack on which the partial results can be placed.

Moving from the type ENV to the type ENS and from N to STK are classic examples of generalisation, which we discussed in our elementary example in the previous section. In these cases the embedding takes each environment to a singleton ensemble and each number to a singleton stack.

If we introduce stacks of partial results and environments we will need to consider how this affects the types of certain continuations. In fact those for declarations can remain the same, since a declaration generates an environment and not an ensemble. On the other hand the continuations for expressions must be generalised (again in the sense of Section 2) so that they accept arguments of type List(N) (= STK) rather than just N by setting ECON = STK $\rightarrow S \rightarrow S$. Moreover, the function that evaluates expressions will be extended to take a stack as an explicit argument.

We need a type simulation for the next continuations in terms of the old. In fact we can introduce a representation function for the new continuations for expressions and the partial result stack in terms of the old, and for ensembles in terms of environments. To this end we define:

Repecont $k\,z = \lambda n.\,k(n:z)$

The intuition here is that, in the old version, partial results are embedded within the continuations, whereas, in the new version, they appear explicitly as a stack. **Repecont** embeds the stack in the appropriate fashion.

Repens y = **smooth** y

where **smooth** = **reduce append** [ ]. That is, **smooth** is **append** extended to lists of lists.

Armed with these we can provide eureka definitions for three new functions of the following types:

Cmdd $\in$ CMD $\rightarrow$ ENS $\rightarrow$ CCON $\rightarrow S \rightarrow S$

Expp $\in$ EXP $\rightarrow$ ENS $\rightarrow$ ECON $\rightarrow$ STK $\rightarrow S \rightarrow S$

Decc $\in$ DEC $\rightarrow$ ENS $\rightarrow$ DCDN $\rightarrow S \rightarrow S$

Cmdd $c\,y\,q\,s$ = **Cmd** c (**Repens** y) $q\,s$

Expp $e\,y\,k\,z\,s$ = **Exp** e (**Repens** y (**Repecont** $k\,z$) s

Decc $d\,y\,x\,s$ = **Dec** d (**Repens** y) $x\,s$

It is necessary to define the following function for linking environments and ensembles:

$p \Delta [\,] = [p]$

$p \Delta (p' : y)$ = (**append** $p\,p'$) : y

After a straightforward transformation, again utilising the natural substructure ordering on the data, we obtain:

Cmdd $(i := e)\,y\,q\,s$ = **Expp** $e\,y$ (**ASSIGN** $i\,y\,d\,q$) [ ] s

Cmmd $(c; c')\,y\,q\,s$ = **Cmdd** $c\,y$ (**Cmdd** $c'\,y\,q$) s

Cmdd (begin d; c end) $y\,q\,s$ =
Decc $d\,y(\lambda p'.$ **Cmdd** $c\,(p \Delta([\,] : y))\,q)\,s$

Expp $n\,y\,k\,z\,s = k(n:z)\,s$

Expp $i\,y\,k\,z\,s = k((s(y\,i)):z)\,s$

Expp $(e + e')\,y\,k\,z\,s$ = **Expp** $e\,y$ (**Expp** $e'\,y$(**PLUS** k)) $z\,s$

Decc (var i) $y\,x\,s$ = **DECL** $i\,y\,x\,s$

Decc $(d; d')\,y\,x\,s$ =

Decc $d\,y(\lambda p.$ **Decc** $d'(p \Delta y)(\lambda p'.\,x(p' \Delta (p \Delta y))))\,s$

for suitable simple redefinitions of the functions **ASSIGN**, **DECL** and so on.

Note how there is no lambda abstraction over values in the function **Expp**. This is because values have been factored out of the continuations. On the other hand we still have abstraction over environments in **Decc**. Factoring out these, which we now turn to, is more difficult because of the multiple threading of environments which occurs, particularly, in the second equation for commands.

Inspection of the types shows that ensembles are embedded within continuations. A typical continuation, say: (**Cmdd** $c\,y\,q$) is already relativised with respect to y. Note that it is not relativised with respect to s or z. We are led to define a new type of continuations called universal continuations, which factor out all the data types we are using:

$u \in$ UCON = ENS $\rightarrow$ STK $\rightarrow S \rightarrow S$.

In order to underline their universality we define the type of all statements:

$\pi \in \mathrm{STM} = \mathrm{CMD} + \mathrm{EXP} + \mathrm{DEC}$

and the type of a single, proposed function:

$\mathbf{Eval} \in \mathrm{STM} \to \mathrm{UCON} \to \mathrm{UCON}.$

This can be unrolled to:

$\mathbf{Eval} \in \mathrm{STM} \to \mathrm{UCON} \to \mathrm{ENS} \to \mathrm{STK} \to S \to S$

and this demonstrates that statements are evaluated with respect to an ensemble, a stack, a store and a universal continuation (representing the rest of the execution without reference to any values of ENS, STK or S).

We will need to specify constraints on this new function in order to undertake a transformation. This presupposes that we have at our disposal notions of simulation between a number of higher-order types. For convenience we define:

$t \in \mathrm{TUC} = \mathrm{UCON} \to \mathrm{UCON}$

the type of transformations of universal continuations. We wish to know when an element $t \in \mathrm{TUC}$ simulates a value $m \in \mathrm{CM}$, $r \in \mathrm{EM}$ or $v \in \mathrm{DM}$. Furthermore, we will need to know when a universal continuation, $u \in \mathrm{UCON}$, simulates a value $q \in \mathrm{CCON}$, $k \in \mathrm{ECON}$ or $x \in \mathrm{DCON}$.

Let us start with the type CCON. Three points stand out: elements of CCON are relativised with respect to ENS, they are not relativised with respect to the type S and they are not defined over the type STK. Elements of UCON are defined over all three types but are not relativised with respect to any of them. We are led to define a three-place relation written:

$(\_ \approx \_) \bmod \_ \subseteq \mathrm{UCON} \times \mathrm{CCON} \times \mathrm{ENS},$

which relates elements of UCON and CCON modulo an ensemble. We set:

$(u \approx q) \bmod y \Leftrightarrow_{\mathrm{def}} (\forall z \in \mathrm{STK})((u\,y\,z) = q)$

We will use the same notation for relations of other types. The context will always indicate which relation is intended.

Elements of ECON are also relativised with respect to an ensemble. This leads to:

$(u \approx k) \bmod y \Leftrightarrow_{\mathrm{def}} (u\,y) = k.$

The domain of the type DCON is ENV (and not ENS). These elements are partially relativised with respect to ensembles. Note that DCON is not defined over the type STK. This leads us to:

$$(u \approx x) \bmod y \Leftrightarrow (\forall z \in \mathrm{STK})(\forall p \in \mathrm{ENV})((u(p\,\Delta\,y)\,z) = (x\,p)).$$

Next we move to the higher-order types. These are quite straightforward and ensure that continuation transformers preserve the representations defined above:

$$(t \approx m) \Leftrightarrow_{\mathrm{def}} (\forall y, q, u)((u \approx q) \bmod y \Rightarrow ((t\,u) \approx (m\,y\,q)) \bmod y)$$

$$(t \approx r) \Leftrightarrow_{\mathrm{def}} (\forall y, k, u)((u \approx k) \bmod y \Rightarrow ((t\,u) \approx (r\,y\,k)) \bmod y)$$

$$(t \approx v) \Leftrightarrow_{\mathrm{def}} (\forall y, x, u)(u \approx x) \bmod y \Rightarrow ((t\,u) \approx (v\,y\,x)) \bmod y)$$

We are now in a position to specify the relational constraints we wish to impose in order to characterise a new function **Eval**:

$(\mathbf{Eval}\,c) \approx (\mathbf{Cmdd}\,c)$

$(\mathbf{Eval}\,e) \approx (\mathbf{Expp}\,e)$

$(\mathbf{Eval}\,d) \approx (\mathbf{Decc}\,d)$

Some example steps in the transformation are the following.

Let $(u \approx q) \bmod y$.
$\mathbf{Eval}\,(c;c')\,u\,y\,z =$ (for any z)
$\mathbf{Cmdd}\,(c;c')\,y\,q =$
$\mathbf{Cmdd}\,c\,y\,(\mathbf{Cmdd}\,c'\,y\,q) =$
[from the assumption we can fold to obtain:
$((\mathbf{Eval}\,c'u) \approx (\mathbf{Cmdd}\,c'\,y\,d)) \bmod y$
and by folding again we have:
$((\mathbf{Eval}\,c\,(\mathbf{Eval}\,c'u)) \approx (\mathbf{Cmdd}\,c\,y\,(\mathbf{Cmdd}\,c'\,y\,q))) \bmod y$]
$\mathbf{Eval}\,c\,(\mathbf{Eval}\,c'u)\,y\,z$

Note that it is the substructure well-ordering which we appeal to in this transformation. Next we tackle the most difficult step, for blocks. We will need two functions of type TUC which we alluded to earlier:

$\mathbf{OLD} \in \mathrm{UCON} \to \mathrm{UCON}$
$\mathbf{NEW} \in \mathrm{UCON} \to \mathrm{UCON}$

$\mathbf{OLD}\,u(p:y)\,z\,s = u\,y\,z\,s$

$\mathbf{OLD}\,u[\,]\,z\,s = u[\,]\,z\,s$

$\mathbf{NEW}\,u\,y\,z\,s = u([\,]:y)\,z\,s$

Let $(u = q) \bmod y$.
$\mathbf{Eval}\ (\mathrm{begin}\,d;c\ \mathrm{end})\,u\,y\,z =$ (for any z)
$\mathbf{Cmdd}\ (\mathrm{begin}\,d;c\ \mathrm{end})\,y\,q =$
$\mathbf{Decc}\,d\,y(\lambda p.\,\mathbf{Cmdd}\,c\,(p\Delta([\,]:y))\,q =$
[from the assumption we can fold to obtain:
$((\mathbf{Eval}\,c\,(\mathbf{OLD}\,u)) \approx$
$(\mathbf{Cmdd}\,c\,(p\Delta([\,]:y))\,q)) \bmod (p\Delta([\,]:y))$
This is subtle: note that u and q are related modulo y so we need to arrange that the ensemble supplied to u is y, and not $p':y$ for some $p' \in \mathrm{ENV}$. The above holds for any $p \in \mathrm{ENV}$, so we can use the relationship between UCON and DCON, yielding:
$(\mathbf{Eval}\,c\ (\mathbf{OLD}\,u)) \approx (\lambda p.\,\mathbf{Cmdd}\,c(p\Delta([\,]:y))\,q) \bmod [\,]:y$
Folding again provides us with:
$(\mathbf{Eval}\,d(\mathbf{Eval}\,c\ (\mathbf{OLD}\,u))) \approx$
$(\mathbf{Decc}\,d\,y\,(\mathbf{Cmdd}\,c\,(p\Delta([\,]:p))\,q)) \bmod [\,]:y]$
$\mathbf{NEW}\,(\mathbf{Eval}\,d(\mathbf{Eval}\,c\,(\mathbf{OLD}\,u)))\,y\,z\,s$

For an example of a transformation step for a primitive function let us take **PLUS**. We need a function **Plus** of type TUC. The constraint is established via the relationship between TUC and EM. Assume that we have $(u \approx k) \bmod y$.

$\mathbf{Plus}\,u\,y\,(n:n':z) =$
$\mathbf{PLUS}\,k(n:n':z) =$ (by assumption)
$\mathbf{PLUS}\,(u\,y)\ (n:n':z) =$ (by definition) $u\,y\ ((n+n'):z)$.
Thus we have

$\mathbf{Plus}\,u\,y(n:n':z)\,s = u\,y((n+n'):z)\,s$

It is now possible to rewrite the result of the transformation using a single connective defined as follows:

$t \to t' = \lambda uyzs.\,t(t'u)\,yzs$

Proposition 3.3

$\rightarrow$ is an associative operation. ■

Eval $(i := e)$ = **(Eval** e**)** $\rightarrow$ **Assign**

Eval $(c; c')$ = **(Eval** c**)** $\rightarrow$ **(Eval** c'**)**

Eval (begin $d; c$ end) =

NEW $\rightarrow$ **(Eval** d**)** $\rightarrow$ **(Eval** c**)** $\rightarrow$ **OLD**

Eval n = **Push** n

Eval i = **Lookup** i

Eval $(e + e')$ = **(Eval** e**)** $\rightarrow$ **(Eval** e'**)** $\rightarrow$ **Plus**

Eval (var i) = **Decl** i

Eval $(d; d')$ = **(Eval** d**)** $\rightarrow$ **(Eval** d'**)**

This looks rather like a code generator. Note that the entities which are manipulated by the connective are continuations of continuations. Indeed, there is no reason why this formulation should not be concretised by a final transformation as follows:

We begin by defining an enumerated type of instructions (INS) by means of:

$\tau \in$ INS = {new, old, lookup i, decl i, push n, assign, plus}

for all $i \in$ VAR and $n \in$ NUM. From this we define a type UCOM of universal completions. Such first-order representatives of continuations were introduced in Ref. 5.

$\delta \in$ UCOM = List (INS)

Now we can introduce a new evaluation function, **eval**, as follows:

eval $\in$ (STM $\times$ UCOM $\times$ ENS $\times$ STK $\times$ S) $\rightarrow$ S

given by

eval $\langle \pi, \delta, y, z, s \rangle$ = **Eval** π **(rep** δ**)** $y\, z\, s$

where

rep $\in$ UCOM $\rightarrow$ UCON

rep [ ] = **id**

rep $(\tau : \varsigma)$ = $[\tau]$ **(rep** δ**)** and

[new] = NEW and so on for other members of INS.

The details of the transformations are not exciting and we omit them. Note however that the associativity of TUC-composition is a crucial lemma.

Our main motivation, however, was to see how higher-order transformations and type simulations could be used to obtain single-threaded data objects from multi-threaded entities. It seems that in this case we can do so, most significantly for environments but also, in fact, for partial results. In both cases it is interesting to note that we pay the price of complicating the data objects in the factoring. Environments were generalised to ensembles and numbers generalised to stacks (of numbers). In a sense we undertook, rigorously, what a programming language implementer does informally. That is, find a way to couch programs (which are trees) as code (which is a list or sequence) by suitable choices of data structures (probably combined to form some notion of run-time data frames).

REFERENCES

1. R. Burstall and J. Darlington, A transformation system for developing recursive programs. *J. ACM*, **24** (1) (1977)
2. J. Darlington and R. Burstall, A system which automatically improves programs. *Acta Informatica* **6**, 41–60 (1976).
3. J. Darlington, A synthesis of several sorting algorithms. *Acta Informatica* **11**, 1–30 (1978).
4. P. Henderson, Functional Programming. Prentice-Hall, Englewood Cliffs, NJ (1980).
5. M. C. Henson and R. Turner, Completion semantics and interpreter generation. *Proceedings 9th ACM Symposium on the Principal Programming Languages, Albuquerque, New Mexico, 1982.*
6. M. C. Henson, *Higher-Order Transformations and Relational Constraints.* Internal report no. 104, Department of Computer Science, University of Essex.
7. J. C. Raoult and R. Sethi, The global storage needs of a subcomputation. *Proc. ACM Symp. POPL*, pp. 148–157 (1983).
8. J. C. Raoult and R. Sethi, *Properties of a Notation for Combining Functions*, Lecture Notes in Computer Science, **140**, 429–441, Springer, (1982).
9. D. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. Prog. Lang. and Sys.* **7**, 299–310 (1985).
10. D. Schmidt, *Denotational Semantics, a Methodology for Program Development.* Allyn and Bacon, Newton, MA (1986).
11. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA (1977).
12. C. Strachey and C. Wadsworth, *Continuations – a Mathematical Semantics for Handing Full Jumps.* Technical Monograph PRG-11, Programming Research group, University of Oxford (1974).
13. D. A. Turner, Miranda: a Non-Strict Functional Language with Polymorphic Types (Proc. Conf. on functional programming and computer architecture), Lecture Notes in Computer Science, **201**, Springer, Heidelberg (1985).
14. M. Wand, Continuation based transformation strategies, *J. ACM*, **27** (1) 164–80 (1980).
15. M. Wand, Semantics directed machine architecture. *Proc. 9th ACM Symp. POPL* (1982).