View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Theoretical Computer Science 93 (1992) 91–113
Elsevier

A new approach to recursion removal

Peter G. Harrison and Hessam Khoshnevisan

Department of Computing, Imperial College of Science, Technology and Medicine, University of London, London SW7 2BZ, UK

Communicated by M. Nivat
Received June 1989
Revised December 1989

Abstract

Harrison, P.G. and H. Khoshnevisan, A new approach to recursion removal, Theoretical Computer Science 93 (1992) 91–113.

Iterative forms are derived for a class of recursive functions, i.e. the recursion is "removed". The transformation comprises first *analysis* of the defining equation of a recursive function and then *synthesis* of an imperative language loop from the primitive subexpressions so obtained. This initially leads to a two-loop program but further transformation provides a single loop version under appropriate conditions. The analysis–synthesis approach contrasts with previous methods using template matching and induces a constructive method which is better suited to mechanisation, although its implementation is not considered here.

1. Introduction

A route to improving the often poor performance of functional programs is to *transform* recursively defined programs into more efficient versions – ideally, tailored to suit the architecture on which they are executed. Thus, for execution on a von Neumann machine, one should aim to derive an equivalent *iterative* solution, i.e. using loops in the PASCAL style, which will optimise both execution time and storage. (This may also benefit parallel architectures by providing a natural mechanism for achieving large-grain parallelism, which many believe is fundamental to the whole issue of concurrent evaluation.)

Our approach to this kind of recursion removal is based upon the ascending Kleene chain (AKC) of the function in question. When successive approximating expressions grow linearly, we show how to construct the required loop from the first one that is sufficient for the argument to which the function is to be applied. The transformation is, therefore, in three parts:

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

- definition of the required properties a function must possess in order to have a linear AKC;
- analysis of a given function's defining expression to determine whether these properties are satisfied and to decompose the expression into primitive subexpressions if so;
- synthesis of an imperative language loop from the primitive subexpressions so obtained.

This initially leads to a two-loop program but further transformation, using specific properties of the primitive functions involved in the defining equation, provides a single *reversed-loop* implementation under appropriate conditions. The analysis–synthesis approach contrasts with previous methods using template matching, e.g. [6, 4, 12], and induces a constructive method which is better suited to mechanisation (although implementation issues are not the subject of the present work). It can also transform some higher-order functions. Our approach increases the generality of previous work to some degree as well as unifying existing schemes into a common framework.

In the next section we give the underlying analysis for the transformation of linear functions and our main result is then presented in Section 3 which derives an equivalent *single* loop for linear functions that satisfy the appropriate conditions. Several detailed examples showing how the results may be applied are also given in Section 3 along with a comparison with related work. In Section 4 the more general transformation into a pair of loops is given and further optimisations are described which may also result in a (different) single loop. The paper concludes in Section 5.

The analysis is presented in terms of a combinator-based language in which there are no variables representing objects in the underlying domain. The primitive combinators include function *composition* (denoted by $\_\circ\_$), *conditional* (denoted $\_\rightarrow\_;\_$) and tupling which we call *construction* (denoted by $[\_,\_\ldots\_,\_]$). The syntax is therefore in the FP style, and we denote function application by a colon and write Hf, synonymously with $H(f)$, to denote the application of the *functional* H to the function f (cf. [2]). However, our results are equally applicable to *any* functional language if we first abstract object variables and remove pattern matching (by forming a conditional tree in FP). We also use the combinators **K** and **APPLY** – which takes a pair as argument and applies the first component (a function) to the second. The combinator **S** can then be defined as (**APPLY** $\circ$ [**APPLY** $\circ$ [1, 3], **APPLY** $\circ$ [2, 3]]) giving full higher-order expressive power.

2. The transformation for general linear functions

The AKC for a function f, defined as the least fixed point of the equation $f = Ff$ (i.e. of the functional F) is $f_0, \ldots, f_n, f_{n+1}, \ldots$, where $f_0 = \underline{\perp}$ and $f_{n+1} = Ff_n$ for $n \geqslant 0$ (we write $\boldsymbol{a}$ or $\underline{a}$ to denote the constant function defined by $\boldsymbol{a}{:}x = a$ for all objects $x \neq \perp$ and $\perp$ if $x = \perp$). Thus, if the result of applying f to an object x is finite, $f{:}x = f_n{:}x$ for some integer

$n \geqslant 0$. For functionals F having an AKC that grows linearly, we will generate an iterative solution for f which first computes this value of n as a function of x and then constructs a for-loop which is executed n times. We restrict ourselves to finite result-objects and also assume that functions are strict so that applicative order evaluation can be used safely. Of course, an attempt to compute $f_\infty{:}x$ in an infinite (for) loop could be regarded as correct in the sense that it would approximate $f{:}x$ ever more closely and lazy evaluation could be used. Unfortunately, however, the computation of n would not terminate and the loop would never be entered in a sequential implementation.

We now have three problems: finding conditions under which an AKC grows linearly and the analysis and synthesis phases of the transformation discussed in the introduction. The first problem has already been solved in [2]. A function has a linearly growing AKC if it is itself *linear*. A linear function f has the definition $f = p \rightarrow q; Hf$ for fixed functions p, q and *linear functional* H. A functional H is linear if it satisfies the property that for all functions a, b, c, $H(a \rightarrow b; c) = H_t a \rightarrow Hb; Hc$ for some functional H_t, called the *predicate transformer* (p.t.) of H. Linear functions satisfy the *linear expansion theorem* (LET) which asserts that in the AKC of f

$$f_n = p \rightarrow q; H_t p \rightarrow Hq; \ldots; H_t^n p \rightarrow H^n q; H^n \perp .$$

From the LET it follows that given object x as argument, if $f{:}x$ is defined and finite, then $f{:}x = (H^n q){:}x$, where n is the least integer such that $(H_t^n p){:}x = T$. Thus, for the application of f to x, f can be "computed" iteratively in a loop on the domain of functions, starting with q in the "accumulator" and applying H to the accumulator n times. Of course, in general, the increasing complexity of the representation of the sequence of functions $q, Hq, H^2 q, \ldots$ renders this approach impractical, and further transformation is needed to derive an equivalent loop at the object level. Note that in a strict language such as FP, if $f{:}x$ is infinite, i.e. an n cannot be found, the result computed for $f{:}x$ will be $\perp$, not necessarily $(\lim_{n \to \infty} H^n \perp){:}x$. However, an additional condition demanded of a linear functional ensures that this limit must also be $\perp$ if $(H_t^n p){:}x = F$ for all n. We adopt the same restriction so that a nonterminating loop is indeed semantically correct.

The idea of the present approach is that if a loop implementation exists for the expression $f{:}x$, the loop should comprise the assignment to an *accumulator* of some expression which depends on the current value in that accumulator. The accumulator will, therefore, be updated successively and eventually hold the required result. More precisely, on the ith iteration, the assigned expression is a function of two variables, the current value of a loop input variable, x_i (given by the loop count i), and the value of the previous loop result r_{i-1} (the accumulator).

2.1. Formal analysis

Following the approach outlined above, the loop equivalent to a linear function is given by the propositions which follow. Proposition 2.1, which is essentially taken

from [2], classifies various linear functionals and gives their predicate transformers.

Proposition 2.1. *For function variables f, b and fixed functions a, p, $a_1,\ldots,a_n (n\geqslant 1)$, the functional H is linear with predicate transformer H_t if*

(a) $Hf=a$ *with* $H_t b=\boldsymbol{T}$,
(b) $Hf=f\circ a$ *with* $H_t b=b\circ a$,
(c) $Hf=a\circ f$ *with* $H_t b=b$,
(d) $Hf=[a,f]$ *or* $[f,a]$ *with* $H_t b=b$,
(e) $Hf=p\rightarrow Af; Bf$ *where* A, B *are linear, with* $H_t b=p\rightarrow A_t b; B_t b$,
(f) $H=AB$ *where* A, B *are linear, with* $H_t=A_t B_t$,
(g) $Hf=[g_1, g_2,\ldots,g_n]$ *and for* $1\leqslant i\leqslant n$,
  (i) $g_i=a_i$ *or* $g_i=H_i f$, *where* H_i *is linear*,
  (ii) *for* $1\leqslant j, k\leqslant n$ *if* $g_j=H_j f$ *and* $g_k=H_k f$ *then* $H_{jt}=H_{kt}$; *then the p.t.* $H_t=H_{jt}$ *for any* j *s.t.* $g_j=H_j f\ (1\leqslant j\leqslant n)$, *and* $H_t b=\boldsymbol{T}$ *if no such* j *exists.*
(h) $Hf=Pf\rightarrow Af; Bf$, *where* P, A *and* B *are linear and* $P_t=A_t=B_t$; *then* $H_t=P_t=A_t=B_t$.

Note that (g) subsumes (d) and reduces to (a) if $g_i=a_i$ for all i, $1\leqslant i\leqslant n$. The proofs of (a)–(g) may be found in [2]. We prove (h) as follows.

Proof of Proposition 2.1(h). If $Hf=Pf\rightarrow Af; Bf$ and $P_t=A_t=B_t$, then

$$\begin{aligned} H(a\rightarrow b; c) &= (P_t a\rightarrow Pb; Pc)\rightarrow(A_t a\rightarrow Ab; Ac); (B_t a\rightarrow Bb; Bc) \\ &= P_t a\rightarrow(Pb\rightarrow Ab; Bb); (Pc\rightarrow Ac; Bc) \quad \text{(because } P_t=A_t=B_t\text{)} \\ &= P_t a\rightarrow Hb; Hc. \end{aligned}$$

Therefore, H is linear with $H_t=P_t$. □

All linear functions satisfy the *functional composition theorem* [2] which states that if H and G are linear functionals with predicate transformers H_t and G_t respectively, then their composition HG, defined by $(HG)f=H(Gf)$ for function variable f, is also linear with predicate transformer $H_t G_t$. We consider functions which are least fixed points of equations of the form $f=p\rightarrow q; Hf$, where the functional H is in the class of linear functionals defined inductively as follows.

Definition 2.2. H is a composite linear form (CLF) if

(a) $Hf=f$, or
(b) $Hf=(Gf)\circ a$, where G is a CLF, or
(c) $Hf=a\circ(Gf)$, where G is a CLF, or
(d) $Hf=[g_1, g_2,\ldots, g_n]$ and, for $1\leqslant i\leqslant n$.
  (i) $g_i=a_i$ or $g_i=H_i f$, where H_i is a CLF with p.t. H_{it}, the same for all such i,
  (ii) $g_j=H_j f$ for at least one j $(1\leqslant j\leqslant n)$, or

(e) $Hf = p \to Af; Bf$, where A, B are CLFs, or

(f) $Hf = Pf \to Af; Bf$, where P, A, B are CLFs and $P_t = A_t = B_t$.

Note. In case (d)(ii), if we were to allow $g_i = a_i$ for all $1 \leqslant i \leqslant n$, then H would be a fixed functional. The function defined by $f = p \to q; Hf$ would then also be fixed, i.e. not even recursive. Henceforth, the condition will be assumed to hold implicitly.

In other words we will look at functions with defining equations given in terms of only the three combining forms, composition, construction and conditional. This hierarchical definition encourages a parser-based transformation system.

The predicate transformer of any CLF now follows immediately.

Proposition 2.3. *For function variable b, using the corresponding labelling of Definition 2.2, the CLF H is a linear functional with predicate transformer:*

(a) $H_t b = b$,

(b) $H_t b = (G_t b) \circ a$,

(c) $H_t b = G_t b$,

(d) $H_t = H_{jt}$ *for j such that* $g_j = H_j f$,

(e) $H_t b = p \to A_t b; B_t b$,

(f) $H_t = P_t = A_t = B_t$.

Proof. The proof is straightforward and uses the results of Proposition 2.1 together with the functional composition theorem.

Furthermore, all CLFs satisfy an important property given by the following proposition.

Proposition 2.4. (proof in Appendix A). *Given CLF H,* $H_t f = f \circ H_t \mathrm{id}$ *for all functions f.*

We now define the loop expression which updates the accumulator corresponding to a CLF H. It is the inductive nature of this definition which facilitates the systematic synthesis of the target loop and which distinguishes our approach from previous ones.

Definition 2.5. Given CLF H, define object-expressions $E_H(u, v)$ as follows:

(a) if $Hf = f$ then $E_H(u, v) = u$,

(b) if $Hf = (Gf) \circ a$ then $E_H(u, v) = E_G(u, a{:}v)$,

(c) if $Hf = a \circ (Gf)$ then $E_H(u, v) = a{:}E_G(u, v)$,

(d) if $Hf = [g_1, g_2, \ldots, g_n]$ and for $1 \leqslant i \leqslant n$, $g_i = a_i$ or $g_i = H_i f$, where H_i is a CLF with p.t. H_t, then $E_H(u, v) = \langle F_1(u, v), \ldots, F_n(u, v) \rangle$, where

$$F_i(u, v) = \begin{cases} a_i{:}v & \text{if } g_i = a_i, \\ E_{\mathrm{Hi}}(u, v) & \text{if } g_i = H_i f \end{cases}$$

(e.g. if $Hf=[H_1 f, H_2 f, \ldots, H_n f]$, where $H_1, \ldots, H_n$ are all CLFs s.t.

$H_{1t}= \cdots = H_{mt}=H_t$, then

$E_H(u, v)=\langle E_{H1}(u, v), E_{H2}(u, v), \ldots, E_{Hn}(u, v)\rangle$),

(e) if $Hf=p\rightarrow Af; Bf$, where A, B are CLFs, then

$E_H(u, v)=\underline{\text{if}}\ p{:}v=T\ \underline{\text{then}}\ E_A(u, v)\ \underline{\text{else}}\ E_B(u, v)$,

(f) if $Hf=Pf\rightarrow Af; Bf$, where P, A and B are CLFs s.t. $P_t=A_t=B_t$, then

$E_H(u, v)=\underline{\text{if}}\ E_P(u, v)\ \underline{\text{then}}\ E_A(u, v)\ \underline{\text{else}}\ E_B(u, v)$.

E_H is, therefore, a function with pairs for its domain. Its definition is given in full as a FP function in Appendix B.

The iterative implementation of a linear function given by a CLF is determined by Theorem 2.7. To prove this theorem we need a further property of CLFs which is given in the following proposition.

Proposition 2.6. (proof in Appendix C). *For CLF H, $Hf{:}x=E_H((H_t f){:}x, x)$ for all functions f and objects x.*

Theorem 2.7. *For CLF H, integer $m\geqslant 0$ and objects x_m, r_0, let the objects x_{i-1} and r_i for $1\leqslant i\leqslant m$ be defined by $x_{i-1}=(H_t\mathrm{id}){:}x_i$ and $r_i=(H^i q){:}x_i$. Then $r_i=E_H(r_{i-1}, x_i)$.*

Proof. Abbreviating E_H to E,

$$
\begin{aligned}
r_i &= (H^i q){:}x_i \\
&= (H(H^{i-1} q)){:}x_i \\
&= E((H_t H^{i-1} q){:}x_i, x_i) && \text{(by Proposition 2.6)} \\
&= E((H^{i-1} q \circ H_t\mathrm{id}){:}x_i, x_i) && \text{(by Proposition 2.4)} \\
&= E((H^{i-1} q){:}x_{i-1}, x_i) && \text{(by definition of } x_{i-1}) \\
&= E(r_{i-1}, x_i) && \text{(by definition of } r_{i-1}).
\end{aligned}
$$

□

Theorem 2.7, together with Proposition 2.4 give us all we need to synthesise a pair of object-level loops, as we describe in Section 4. Although applicable to all linear functions defined by CLFs, this transformation does not always produce a code which is much more efficient than explicit recursion. However, the preceding analysis provides us with foundations upon which to build transformations into a single loop. This produces a substantial optimisation in space as well as in time since it is no longer necessary to store loop input variables. We consider this next.

3. Loop reversal

One way of evaluating the application of a linear function in a *single* loop (or transforming the function into a single-tail recursive function) is to execute the loop from "the other end", i.e. beginning the iteration on the argument rather than on the base case value x_0. This technique, when applicable, is often called loop reversal. The idea is not new (see [4] for a full discussion) but here the reversed loop is synthesised constructively after the analysis phase, avoiding the introduction of unspecified auxiliary functions with appropriate properties. Intuitively, our reasoning is as follows. Instead of the expression E_H and sequence $\{r_i | 0 \leqslant i \leqslant n\}$, we find an expression E'_H and generate a sequence $\{a_j | 0 \leqslant j \leqslant n\}$, with $a_n = r_n = f{:}x$. Writing $y_i = x_{n-i}$ $(0 \leqslant i \leqslant n)$, the sequence $a_0, a_1, \ldots, a_n$ is given by the choice of a_0 together with the relationship

$$a_j = E'_H(a_{j-1}, y_{j-1}) \quad \text{for} \quad 1 \leqslant j \leqslant n.$$

Now, by definition of $\{x_i | 1 \leqslant i < n\}$, $y_j = H_t\mathrm{id}{:}y_{j-1}$ $(1 \leqslant j \leqslant n)$ and $y_0 = x_n = x$ (the argument value). Thus, the sequence $y_0, y_1, \ldots$ can be computed successively since y_0 and $H_t\mathrm{id}$ are known; contrast the sequence $x_0, \ldots, x_n$ which requires x_0 and $(H_t\mathrm{id})^{-1}$. All we need to do is find E'_H and the starting value for the accumulator, a_0. These ideas are formalised in Theorem 3.1 and Corollary 3.2, wherein we specify particular a_0 and E'_H together with sufficient conditions for the loop reversal to be valid.

Theorem 3.1. *Let the function f be defined by $f = p \to q$; Hf for fixed functions p, q and CLF H. Suppose there exist dyadic operators A and E' such that for all objects u, v and w*

$$A(E_H(u, v), w) = A(u, E'(v, w)).$$

Let $y_0 = x$, $y_j = H_t\mathrm{id}{:}y_{j-1}$ and $a_j = E'(y_{j-1}, a_{j-1})$ $(j \geqslant 1)$, where a_0 is a right unit of A. Then $f{:}x = A(q{:}y_n, a_n)$, where $n = \min_i\{H_t^i p{:}x = T\}$. ($d$ is a right unit of a dyadic operator D if $D(z, d) = z$ for all z.)

Proof. We show that $A(r_i, a_{n-i}) = r_n$ for $0 \leqslant i \leqslant n$, in the notation of Theorem 2.7. Now, for $0 \leqslant i \leqslant n$,

$$\begin{aligned}
A(r_i, a_{n-i}) &= A(r_i, E'(y_{n-i-1}, a_{n-i-1})) && \text{(by definition of } a_i\text{)} \\
&= A(E_H(r_i, y_{n-i-1}), a_{n-i-1}) && \text{(by hypothesis)} \\
&= A(E_H(r_i, x_{i+1}), a_{n-i-1}) && \text{(since } y_{n-i-1} = x_{i+1}\text{)} \\
&= A(r_{i+1}, a_{n-i-1}) && \text{(by definition of } r_{i+1}\text{)} \\
&= A(r_n, a_0) && \text{(by repeated application)} \\
&= r_n && \text{(since } a_0 \text{ is a right unit of } A\text{).}
\end{aligned}$$

Thus, taking $i=0$,

$$r_n = A(r_0, a_n)$$
$$= A(q{:}y_n, a_n) \quad \text{(since } r_0 = q{:}x_0 \text{ and } x_0 = y_n\text{)} \qquad \square$$

Corollary 3.2. *The theorem holds if*:

(1) E_H *is associative and we choose* $A = E' = E_H$, *or*
(2) $Hf = h \circ Gf$, h *is associative,* $E_G(u, v) = \langle u, \delta(v)\rangle$ *for some function* δ *and we choose* $A = h$, $E' = h \circ [2,1] \circ E_G \circ [2,1]$ (i.e. $E'(u, v) = h(\delta(u), v)$), *or*
(3) $Hf = h \circ Gf$, h *is associative,* $E_G(u, v) = \langle \delta(v), u\rangle$ *for some function* δ *and we choose* $A = h \circ [2,1]$, $E' = h \circ [2,1] \circ E_G \circ [2,1]$ (i.e. $E'(u, v) = h(v, \delta(u))$).

Proof.

(1)
$$A(u, E'(v, w)) = E_H(u, E_H(v, w)) \quad \text{(by definition of } A \text{ and } E'\text{)}$$
$$= E_H(E_H(u, v), w) \quad \text{(since } E_H \text{ is associative)}$$
$$= A(E_H(u, v), w) \quad \text{(since } A = E_H\text{)}$$

(2)
$$A(u, E'(v, w)) = h(u, h(\delta(v), w)) \quad \text{(by definition of } A \text{ and } E'\text{)}$$
$$= h(h(u, \delta(v)), w) \quad \text{(since } h \text{ is associative)}$$
$$= h(h{:}E_G(u, v), w) \quad \text{(by hypothesis of the Corollary 3.2(2))}$$
$$= h(E_H(u, v), w) \quad \text{(by case (c) of Definition 2.5)}$$
$$= A(E_H(u, v), w) \quad \text{(since } A = h\text{)}$$

(3)
$$A(u, E'(v, w)) = h(h(w, \delta(v)), u) \quad \text{(by definition of } A \text{ and } E'\text{)}$$
$$= h(w, h(\delta(v), u)) \quad \text{(since } h \text{ is associative)}$$
$$= h(w, h{:}E_G(u, v)) \quad \text{(by hypothesis of case (3) of Corollary 3.2)}$$
$$= A(h{:}E_G(u, v), w) \quad \text{(by definition of } A\text{)}$$
$$= A(E_H(u, v), w) \quad \text{(by case (c) of Definition 2.5)} \qquad \square$$

For example, the factorial function satisfies the conditions of cases (1) and (2) of the corollary and the definition of the reverse function given in the next section satisfies the conditions of Case (2).

We can now state the main optimising transformation of this section, which is based on the results of Theorem 3.1. For function f satisfying the conditions of Theorem 3.1,

the following instantiation-template computes $f{:}x$ for object x:

$r{:=}a_0$;
while $p{:}x \neq \mathrm{T}$ **do**
  begin
    $r{:=}E'(x, r)$;
    $x{:=}H_t\mathrm{id}{:}x$;
  end
$r{:=}A(q{:}x, r)$;

In particular, for a function f satisfying the conditions of case (2) of Corollary 3.2 the following instantiation-template computes $f{:}x$ for object x:

$r{:=}a_0$;
while $p{:}x \neq T$ **do**
  begin
    $r{:=}h{:}\langle\delta(x), r\rangle$;
    $x{:=}H_t\mathrm{id}{:}x$;
  end
$r{:=}h(q{:}x, r)$;

(Similarly, for a function f satisfying the conditions of case (3) of Corollary 3.2 the template will be identical to the above with the accumulator-updating instructions replaced by $r{:=}h{:}\langle r, \delta(x)\rangle$; and $r{:=}h(r, q{:}x)$;.)

The equivalent tail-recursive forms for the function f corresponding to these templates are:

$$f{=}p{\to}q;\ A\circ[q\circ 1, 2]\circ f'\circ[\mathrm{id}, \boldsymbol{a}_0],\quad f'{=}p\circ 1{\to}\mathrm{id}; f'\circ[H_t\mathrm{id}\circ 1, E'],$$

and

$$f{=}p{\to}q;\ h\circ[q\circ 1, 2]\circ f'\circ[\mathrm{id}, \boldsymbol{a}_0],\ f'{=}p\circ 1{\to}\mathrm{id}; f'\circ[H_t\mathrm{id}\circ 1, h\circ[\delta\circ 1, 2]],$$

3.1. Examples of reversed loops

We illustrate the technique first with the factorial function.

Example 1. The factorial function

$$\mathrm{factorial}{=}\mathrm{eq0}{\to}\mathbf{1}; *\circ[\mathrm{factorial}\circ\mathrm{sub}\,1, \mathrm{id}].$$

The else part $Hf{=}*\circ[f\circ\mathrm{sub}\,1, \mathrm{id}]$ is linear and for all functions g, we may write

$$Hg{=}*\circ G_1 g,$$

where $G_1 g{=}[G_2 g, \mathrm{id}]$, $G_2 g{=}G_3 g\circ\mathrm{sub}1$, $G_3 g{=}g$.

Thus,

$$
\begin{aligned}
E_H &= * \circ E_{G1} && \text{(by case (c) of Definition 2.5)}\\
&= * \circ [E_{G2}, \mathrm{id}{:}2] && \text{(by case (d) of Definition 2.5)}\\
&= * \circ [E_{G3} \circ [1, \mathrm{sub}\,1 \circ 2], 2] && \text{(by case (b) of Definition 2.5)}\\
&= * \circ [1, 2] && \text{(by case (a) of Definition 2.5)}\\
&= *, &&\\
H_t\,\mathrm{id} &= \mathrm{sub}\,1 && \text{(by Proposition 2.3).}
\end{aligned}
$$

Therefore, to reverse the loop, we need only that multiplication is associative (to satisfy the condition of case (1) of Corollary 3.2) and to know a right unit for it. Thus, we can indeed compute factorial "from the top" in its above definition.

Thus, by case (1) of Corollary 3.2 we can generate the following imperative code for the factorial function:

```
r:= 1;              (* since 1 is a right unit for multiplication *)
while eq0:x ≠ T do
      begin
        r:= *:⟨x, r⟩;
        x:= sub 1:x;
      end
r:= *:⟨1, r⟩;       (* this instruction is in fact redundant *)
```

Factorial also satisfies the conditions of case (2) of the corollary since multiplication is associative and $\delta = \mathrm{id}$.

Example 2. The reverse function rev

Consider the following definition of reverse function in terms of the primitive function append.

$$\mathrm{rev} = \mathrm{null} \rightarrow [\,]; \mathrm{append} \circ [\mathrm{rev} \circ \mathrm{tl}, [\mathrm{hd}]]$$

The else part $Hf = \mathrm{append} \circ [f \circ \mathrm{tl}, [\mathrm{hd}]]$ is linear and, for all functions g, we may write

$$Hg = \mathrm{append} \circ G_1 g,$$

where $G_1 g = [G_2 g, [\mathrm{hd}]]$, $G_2 g = G_3 g \circ \mathrm{tl}$, $G_3 g = g$.
Thus,

$$
\begin{aligned}
E_H &= \mathrm{append} \circ E_{G1} && \text{(by case (c) of Definition 2.5)}\\
&= \mathrm{append} \circ [E_{G2}, [\mathrm{hd}] \circ 2] && \text{(by case (d) of Definition 2.5)}\\
&= \mathrm{append} \circ [E_{G3} \circ [1, \mathrm{tl} \circ 2], [\mathrm{hd}] \circ 2] && \text{(by case (b) of Definition 2.5)}\\
&= \mathrm{append} \circ [1, [\mathrm{hd}] \circ 2] && \text{(by case (a) of Definition 2.5),}\\
H_t\,\mathrm{id} &= \mathrm{tl}. &&
\end{aligned}
$$

This definition of the function reverse satisfies the conditions of case (2) of the corollary since append is associative and $\delta = [\mathrm{hd}]$. Thus, we can generate the following

imperative code:

```
r:=⟨ ⟩;                          (* since ⟨ ⟩ is a right unit for append *)
while null:x≠T do
      begin
        r:=append:⟨[hd]:x, r⟩;
        x:=tl:x;
      end
r:=append:⟨[ ]:x, r⟩;            (* this instruction is in fact redundant *)
```

or the equivalent tail-recursive form:

$$f = \text{null} \rightarrow [\,]; \text{append} \circ [[\,], 2] \circ f' \circ [\text{id}, [\,]],$$

where $f' = \text{null} \circ 1 \rightarrow \text{id}; f' \circ [\text{tl} \circ 1, \text{append} \circ [[\text{hd}] \circ 1, 2]]$.

Example 3: The list iterator function map

Our techniques can also be applied to some higher-order functions by use of the language primitives **APPLY** and **K**. Consider the "map" higher-order function defined in HOPE as follows:

dec map : list (alpha) × (alpha → beta) → list (beta);

map (nil, *f*) ⇐ nil;

map (*x*::1, *f*) ⇐ [*f*(*x*)] ⟨ ⟩ map (1, *f*);

which we may represent as

$$\text{map} = \text{null} \circ 1 \rightarrow [\,]; \text{append} \circ [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]], \text{map} \circ [\text{tl} \circ 1, 2]]$$

which does not involve **K**. **APPLY** can be taken as a fixed function, and parsing of the above definition yields the following:

In the else part, $Hf = \text{append} \circ [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]], f \circ [\text{tl} \circ 1, 2]]$, is linear and for all functions g, we may write

$$Hg = \text{append} \circ G_1 g,$$

where $G_1 g = [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]], G_2 g]$, $G_2 g = G_3 g \circ [\text{tl} \circ 1, 2]$, $G_3 g = g$.
Thus,

$$\begin{aligned} E_H &= \text{append} \circ E_{G1} && \text{(by case (c) of Definition 2.5)} \\ &= \text{append} \circ [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]] \circ 2, E_{G2}] \\ & && \text{(by case (d) of Definition 2.5)} \\ &= \text{append} \circ [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]] \circ 2, E_{G3} \circ [1, [\text{tl} \circ 1, 2] \circ 2]] \\ & && \text{(by case (b) of Definition 2.5)} \\ &= \text{append} \circ [[\textbf{APPLY} \circ [2, \text{hd} \circ 1]] \circ 2, 1] \\ & && \text{(by case (a) of Definition 2.5),} \end{aligned}$$

$$H_t \text{id} = [\text{tl} \circ 1, 2].$$

This definition of the function map satisfies the conditions of case (3) of the corollary since append is associative and $\delta = [\mathbf{APPLY} \circ [2, \text{hd} \circ 1]]$. Thus, we can generate the following imperative code:

```
r:=⟨ ⟩;                          (* since ⟨ ⟩ is a right unit for append *)
while null:x ≠ T do
      begin
        r:=append:⟨r, [APPLY ∘ [2, hd ∘ 1]]:x⟩;
        x:=[tl ∘ 1, 2]:x;
      end
r:=append:⟨r, [ ]:x⟩;            (* this instruction is in fact redundant *)
```

or the equivalent tail-recursive form:

$$f = \text{null} \rightarrow [\,]; \text{append} \circ [2, [\,], 2] \circ f' \circ [\text{id}, [\,]]$$

where $f' = \text{null} \circ 1 \rightarrow \text{id}; f' \circ [\text{tl} \circ 1, \text{append} \circ [[\text{hd}] \circ 1, 2]]$.

Note that the usual definition of map uses "cons" (i.e. "al") rather than "append", and it is our use of the latter alternative which has permitted the optimisation; since cons is not associative, it is always worthwhile to replace it by append $\circ$ [[1], 2].

Example 4. The list iterator function reduce-right

Consider the "reduce-right" higher-order function defined in HOPE as follows:

```
dec reduce : list(alpha) × (alpha × beta → beta) × beta → beta;

reduce (nil, f, b) ⇐ b;

reduce (x::1, f, b) ⇐ reduce (1, f, f(x, b));
```

which can be represented in our meta-language as

$$\text{reduce} = \text{null} \circ 1 \rightarrow 3; \text{reduce} \circ [\text{tl} \circ 1, 2, \mathbf{APPLY} \circ [2, [\text{hd} \circ 1, 3]]].$$

This definition is already in a tail-recursive form and the derivation of the resultant **while** loop is trivial. However, note that "reduce-left" function defined by

```
dec reduce : list(alpha) × (alpha × beta → beta) × beta → beta;

reduce (nil, f, b) ⇐ b;

reduce (x::1, f, b) ⇐ f(x, reduce(1, f, b));
```

or in the variable-free form by

$$\text{reduce} = \text{null} \circ 1 \to 3;\ \mathbf{APPLY} \circ [2, [\text{hd} \circ 1, \text{reduce} \circ [\text{tl} \circ 1, 2, 3]]]$$

cannot be transformed using Theorem 3.1. This is because the loop expression for this function is given by $\mathbf{APPLY} \circ [2 \circ 2, [\text{hd} \circ 1 \circ 2, 1]]$ and **APPLY** is not associative.

The redundant final instruction in the imperative codes of the examples above may be avoided by recognising that the "then" part of the definition (i.e. q) always generates a unit of A (e.g. 1 in the case where A was $*$ and $\langle\ \rangle$ in the case where A was append).

3.2. Comparison with the related work

In [4], Bauer and Wossner consider a quite extensive set of linear functions which have equivalent "repetitive" (iterative) forms. The transformations they present for linear functions require a priori knowledge about equivalent expression structures which are *assumed* to exist as conditions for their theorems. In contrast, our approach is constructive, being aimed at the systematic synthesis of optimised programs from the decomposed function-defining expression obtained in the analysis phase. Any matching required is very specific and so much simpler, requiring only a few tests (e.g. see Corollary 3.2). Although, in general, such tests are only semi-decidable, e.g. the test for associativity, the same also applies to any comparable scheme. Of course, often the function concerned is primitive whereupon the test is trivial.

Kieburtz and Shultis [12] also operate at the function level using FP, and derive equivalent tail-recursive functions expressed in terms of a "while" (canonical iterative) combining form, for linear functions with certain defining expression structures – again using template matching. They show by fixpoint induction that the function $f = p \to q; h \circ [i, f \circ j]$ is equivalent to $f' = 1 \circ w \circ [g, \text{id}]$ where $w = p \circ 2 \to \text{id}; w \circ [h' \circ [1, i \circ 2], j \circ 2]$. Here if h is associative, then $h' = h$ and g is its constant unit function. A similar result has also been derived in [3], using the linear expansion theorem. It yields a function-level version of Burstall and Darlington's FACTIT transformation [6]: the function w takes as an argument and returns as a result a pair, the first component of which is an accumulator and the second a loop variable. In fact, this result follows immediately as an instance of Theorem 3.1, $Hf = h \circ [i, f \circ j]$ defining H as a CLF, as can be seen by constructing the tail-recursive form of f.

Apart from their constructive nature, the results of this paper are also more general than the above. For example, consider the function definition $f = p \to q; Hf$, where $Hf = h \circ Gf$, h is associative, $Gf = r \to Kf; Lf$, $E_K(u, v) = \langle u, \delta_K(v) \rangle$ and $E_L(u, v) = \langle u, \delta_L(v) \rangle$. This function clearly satisfies the conditions of case (2) of Corollary 3.2 since by case (e) of Definition 2.5 we have

$$\begin{aligned} E_G(u, v) &= r{:}v \to \langle u, \delta_K(v) \rangle; \langle u, \delta_L(v) \rangle \\ &= \langle r{:}v \to u; u, r{:}v \to \delta_K(v); \delta_L(v) \rangle \\ &= \langle u, \delta(v) \rangle \end{aligned}$$

for an appropriately defined δ, provided that r is total or we are using a strongly typed functional language so that r will never produce the result $\perp$. Hence, by application of Theorem 3.2 we can form a single loop for such functions.

An algebraic approach to recursion removal has also been followed by Bird [5], who uses a combination of function-level and object-level equations and reasoning to give a concise presentation which is often ingenious, but intended more as a programming methodology than as a scheme for program transformation. Finally, Chandra [8] considers the space-time trade-offs in the transformation of certain linear functions defined by schemata. He presents a linear-time algorithm and a constant-space algorithm.

4. The general two-loop implementation

Given a linear function definition of the form $f = p \rightarrow q; Hf$, we use Theorem 2.7 and Proposition 2.4 to generate the following iterative code for the expression $f{:}x$ for some object x:

if $p{:}x$
then $r := q{:}x$
else ...

where the "else" part ... (for Hf) will be in two parts, the initialising while-loop followed by the main loop:

```
Loop_Inputs:= PUSH(x, EMPTY_STACK);   (* put x = x_n in the
                                         Loop_Inputs *)
while p:TOS(Loop_Inputs) ≠ T do       (* see note 1 *)
      Loop_Inputs:= PUSH(H_t id:TOS(Loop_Inputs), Loop_Inputs);
r:= q:TOS(Loop_Inputs)                (* since x_0 is now at the head *)
while POP(Loop_Inputs) ≠ EMPTY_STACK do
      begin
        Loop_Inputs:= POP(Loop_Inputs);
        r:= E_H(r, TOS(Loop_Inputs));  (* since x_i is at the head *)
      end
```

Before entering the main loop, the values $\{x_i \mid 0 \leqslant i \leqslant n\}$ are computed – in the general case by a simple **while** loop. This loop is easily constructed using the fact that $n = \min_i \{p{:}\ ((H_t \mathrm{id})^i{:}x) = T\}$ (by Proposition 2.4) in the evaluation of $f{:}x$ (assumed to be defined), and that $x_{i-1} = H_t \mathrm{id}{:}x_i$ for $1 \leqslant i \leqslant n$; H_t, p and x are all known. We know that if the **while** loop is executed n times, in the jth execution, at the top of Loop_Inputs will be x_{n-j} (since it started by having $x = x_n$ in it); so, x_{n-j-1} is simply

computed as $H_t$id:TOS(Loop_Inputs) which is placed at the begining of the list. The condition of the **while** loop ensures that it stops once x_0 is found and inserted. The operations PUSH, POP, TOS and EMPTY_STACK are operations of the target machine and may alternatively be implemented as CONS, TAIL, HEAD and NIL, respectively.

4.1. Source-to-source transformation into two tail-recursive forms

Theorem 4.1. *Given a recursive function f of the form* $f = p \to q; Hf$, *where H is a CLF with predicate transformer* H_t *and loop expression* E_H *(as defined by Definition 2.5 or by its alternative function-level definition given in Appendix B), we have the following equality for the function f:*

$$f = p \to q; f' \circ [q \circ \text{hd}, \text{tl}] \circ \text{list} \circ [\text{id}],$$

where

$$\text{list} = p \circ \text{hd} \to \text{id}; \text{list} \circ \text{al} \circ [H_t\text{id} \circ \text{hd}, \text{id}],$$

$$f' = \text{null} \circ 2 \to 1; f' \circ [E_H \circ [1, \text{hd} \circ 2], \text{tl} \circ 2].$$

(The two tail-recursive functions "list" and f' correspond to the two loops of the code above, the first of which builds up the list of loop inputs, the other corresponding to the main loop.)

Proof. Standard techniques prove that the least fixed points of the equations for f' and "list" yield the least fixed point of the untransformed equation defining $f, f = p \to q; Hf$, when substituted into the above equality. We suggest three methods:

(a) By assuming established equivalences between loops and tail-recursive functions, and applying them to the loop templates derived in the previous section.
(b) By using the LET directly and reasoning at the function-level as outlined in Appendix D.
(c) By fixed point induction, independently of the LET.

Note that in this source-to-source transformation, we are now representing the stack of loop inputs by an ordinary list. Of course, we are also implicitly assuming that tail-recursion is already optimised in some standard way.

In Section 3 we used properties of the fixed functions occurring in the functional H to completely remove the initialisation phase and execute the main loop in reverse order (cf. computing factorial(n) as $n * (n-1) * \cdots * 1$ as opposed to $1 * 2 * \cdots * n$ using the associative property of multiplication). We next consider other optimisations which might be applicable when the techniques of Section 3 cannot be applied. These optimisations may result in the replacement of the first **while** loop by a single expression for the number of iterations n (given by $H_t p$ and the object x), together with expressions for x_0 and for x_i in terms of x_{i-1} in a suitable object domain.

4.2. Avoiding stacks

In the two-loop implementation we must first compute all of the values $\{x_0, x_1, \ldots, x_n\}$. Effectively, the loop implementation has to carry around its own stack. But we can avoid this if x_0 and the function $(H_t\text{id})^{-1}$ are known since x_i can then be computed from x_{i-1} $(1 \leqslant i \leqslant n)$ using the equation $x_i = (H_t\,\text{id})^{-1} : x_{i-1}$ (recall the definition in Theorem 2.7). Hence, there is no need to carry around $\{x_0, x_1, \ldots, x_n\}$ since we can evaluate each x_i in the body of the main loop when it is required – each x_i is used in only one iteration after which it can be discarded. Therefore, a single variable can be used instead of a stack, giving the code

```
n:= 1;
while p:(H_t id:x) ≠ T do
      begin
        n:= n + 1;
        x:= H_t id:x;
      end
r:= q:H_t id:x;
for i:= 1 to n
    do begin
         r:= E_H(r, x);
         x:= (H_t id)^-1 :x;
       end
```

Note, however, that we still need to find x_0. Although some space saving has been made, the increase (or decrease) in execution time will depend on the cost of $(H_t\text{id})^{-1}$ compared to TOS, PUSH and POP. We can optimise the execution time of the above code if it is possible to deduce x_0 from p. For example,

(i) $p = \text{eq0} \Rightarrow x_0 = 0$,

(ii) $p = \text{null} \Rightarrow x_0 = \langle\,\rangle$.

If a linear function is such that an inverse for $H_t\text{id}$ can be constructed and code can be generated to compute x_0, then it is possible to generate code of the form:

```
y:= Base Value;
r:= q:y;
repeat
   y:= (H_t id)^-1 : y;
   r:= E_H(r, y);
until y = x;
```

where the imperative language variable "Base Value" is the value of the expression

generated for x_0. For example, for the factorial function,

```
y:=0;
r:=1;
repeat
   y:=add1:y;
   r:= *:⟨r, y⟩;
until y=x;
```

This is the iterative version of factorial working from the base case *upwards.*

4.3. Examples

Example 1. The transformed FUSC function
The "obfuscate" function [9] is defined by

$$\mathrm{FUSC} = \mathrm{le}\,1 \to \mathrm{id};\ \mathrm{even} \to \mathrm{FUSC} \circ d;\ + \circ [\mathrm{FUSC} \circ d \circ p,\ \mathrm{FUSC} \circ d \circ s],$$

where le 1, d, p and s are the functions less than or equal to 1, divide by 2, add 1 and subtract 1, respectively.

The FUSC function is nonlinear, but the transformation techniques of [11] give an equivalent linear function, namely,

$$\mathrm{FUSC} = 1 \circ g,$$

where

$$g = \mathrm{le}\,1 \to [\mathrm{id}, s];\ \mathrm{le}\,2 \to [Ag, s];\ \mathrm{even} \to [Ag, Dg];\ [Bg, Cg],$$

$$Ag = 1 \circ g \circ d, \quad Bg = + \circ g \circ d \circ p, \quad Cg = 2 \circ g \circ d \circ p \quad \text{and} \quad Dg = + \circ g \circ d.$$

Now we may write $g = \mathrm{le}\,1 \to [\mathrm{id}, s]; Kg$, where $Kg = \mathrm{le}\,2 \to Mg; Hg$, with $Mg = [Ag, s]$, and $Hg = \mathrm{even} \to [Ag, Dg]; [Bg, Cg]$.

We then have (after the usual equational reasoning)

$$E_M(u, v) = \langle 1{:}u,\ s{:}v \rangle \text{ and}$$

$$E_H(u, v) = \underline{\text{if}}\ \mathrm{even}{:}v = T\ \underline{\text{then}}\ \langle E_A(u, v),\ E_D(u, v) \rangle\ \underline{\text{else}}\ \langle E_B(u, v),\ E_C(u, v) \rangle$$

(by case (e) of Definition 2.5)

$$= \underline{\text{if}}\ \mathrm{even}{:}v = T\ \underline{\text{then}}\ \langle 1{:}u,\ +{:}u \rangle\ \underline{\text{else}}\ \langle +{:}u,\ 2{:}u \rangle$$

(by case (d) of Definition 2.5, and that $A_t b = D_t b = b \circ d$,

$B_t b = C_t b = b \circ d \circ p$).

Hence, we have

$$r_i = E_K(r_{i-1}, x_i) = \underline{\text{if}}\ \text{le}\,2{:}x_i = T$$

$$\underline{\text{then}}\ \langle 1{:}r_{i-1}, s{:}x_i \rangle$$

$$\underline{\text{else}}\ (\underline{\text{if}}\ \underline{\text{even}}{:}x_i = T\ \underline{\text{then}}$$

$$\langle 1{:}r_{i-1}, +{:}r_{i-1} \rangle\ \underline{\text{else}}\ \langle +{:}r_{i-1}, 2{:}r_{i-1} \rangle).$$

Example 2. The square root function

Manna and Waldinger [14] transform the specification for the square root of a number given by

$$\text{sqrt}(a, e) = z \quad \underline{\text{iff}} \quad z^2 \leqslant a < (z+e)^2 \quad \underline{\text{where}} \quad 1 \leqslant a \quad \underline{\text{and}} \quad 0 < e$$

(e is the tolerance of the result z with respect to the exact square root of a)

into the function

$$\text{sqrt}(a, e) = \underline{\text{if}} \quad a < e \quad \underline{\text{then}}\ 0$$

$$\underline{\text{else if}} \quad (x+e)^2 \leqslant a \quad \underline{\text{then}}\ x+e$$

$$\underline{\text{else}}\ x,$$

$$\underline{\text{where}} \quad x = \text{sqrt}(a, 2e).$$

The same function written in FP is

$$\text{sqrt}\ \text{lt} \rightarrow \bar{0}; H\text{sqrt},$$

where

$$Hf = \text{le} \circ [\text{square} \circ + \circ [f \circ [1, \text{double} \circ 2], 2], 1] \rightarrow$$

$$+ \circ [f \circ [1, \text{double} \circ 2], 2]; f \circ [1, \text{double} \circ 2].$$

We may write $Hf = Pf \rightarrow Af;\ Bf$ where $Pf = \text{le} \circ [\text{square} \circ + \circ [Gf, 2], 1]$, $Af = + \circ [Gf, 2]$, $Bf = Gf$ and where $Gf = f \circ [1, \text{double} \circ 2]$. We then have, for function p, $P_t p = A_t p = B_t p = p \circ [1, \text{double} \circ 2]$, and for objects u, v $E_B(u, v) = u$, $E_A(u, v) = +{:}\langle u, 2{:}v \rangle$, $E_P(u, v) = \text{le}{:}\langle \text{square}{:}+{:}\langle u, 2{:}v \rangle, 1{:}v \rangle$. Hence,

$$E_H(u, v) = \underline{\text{if}}\ E_P(u, v)\ \underline{\text{then}}\ E_A(u, v)\ \underline{\text{else}}\ E_B(u, v) \quad \text{(by case (f) of Definition 2.5)}$$

$$= \underline{\text{if}}\ \text{le}{:}\langle \text{square}{:}+{:}\langle u, 2{:}v \rangle, 1{:}v \rangle\ \underline{\text{then}}\ +{:}\langle u, 2{:}v \rangle\ \underline{\text{else}}\ u$$

and, so,

$$r_i = \underline{\text{if}}\ \text{le}{:}\langle \text{square}{:}+{:}\langle r_{i-1}, 2{:}x_i \rangle, 1{:}x_i \rangle\ \underline{\text{then}}\ +{:}\langle r_{i-1}, 2{:}x_i \rangle\ \underline{\text{else}}\ r_{i-1}.$$

This loop expression yields the imperative code which implements the following intuitive binary split algorithm (with a simple optimisation outlined in Section 4.2):

```
v:=a;
r:=0;
while e≤v do
            v:=v/2;
            if(r+v)²≤a then r:=r+v;
          od
return r
```

5. Conclusions

We have developed a method that will generate an iterative program for any function defined by a composite linear functional. The technique for the general case is in three parts: detection of a CLF H, determination of its predicate transformer H_t, and construction of its loop expression E_H. This was presented inductively by first recognising the simplest case, $H=\mathrm{ID}$, and forming $H_t=\mathrm{ID}$ and $E_H(u,v)=u$. A more complex CLF is then analysed by detecting compositions, constructions or conditionals applied to simpler CLFs. Proposition 2.3 and Definition 2.5 are then used to build the predicate transformers and loop expressions, respectively, from those of the sub-expressions. The hierarchical decomposition of a function and synthesis of a loop led to the constructive analysis which distinguishes our approach.

Transformation into a single reversed loop utilises Theorem 3.1 and its corollaries, prescribing conditions on the loop expression E_H. If Theorem 3.1 cannot be applied, other enhancements to the resulting two loops may be used which may also generate a (different) single loop. It is also possible to exploit various optimising techniques already developed for imperative languages, e.g. techniques for combining **while** loops. A transformation system based on the techniques presented would clearly be easy to implement in a FP compiler, but equally, all of the results apply to *any* functional programming language such as LISP, HOPE or ML; it is a relatively simple task to first abstract object variables.

Appendix A. Proof of Proposition 2.4

The proof is by structural induction on CLFs in which the base case is covered by case (a) where H is ID. Using the corresponding labelling of Definition 2.2, with the function variable g:

(a) $Hf=f$ (i.e. $H=\mathrm{ID}$), by Proposition 2.3 the p.t. of H is $H_t g=g\circ\mathrm{id}=g\circ H_t\mathrm{id}$.

The inductive step divides into the following five cases.

(b) $Hf=(Gf)\circ a$, where G is a CLF. By Proposition 2.3 we have $H_t g=(G_t g)\circ a=g\circ(G_t\mathrm{id})\circ a$ (by the inductive hypothesis)$=g\circ H_t\mathrm{id}$.

(c) $Hf=a\circ(Gf)$ where G is a CLF. By Proposition 2.4 we have $H_t g=G_t g=g\circ G_t\mathrm{id}$ (by the inductive hypothesis)$=g\circ H_t\mathrm{id}$.

(d) $Hf=[g_1, g_2, \ldots, g_n]$ and for $1\leqslant i\leqslant n$, $g_i=a_i$ or $g_i=H_i f$, where H_i is a CLF with p.t. H_t. By Proposition 2.4 there is a j such that $H_t g=H_{jt}g=g\circ H_{jt}\mathrm{id}$ (by the inductive hypothesis)$=g\circ H_t\mathrm{id}$.

(e) $Hf=p\rightarrow Af; Bf$, where A, B are CLFs. By Proposition 2.4 we have

$$H_t g=p\rightarrow A_t g; B_t g$$

$$=p\rightarrow g\circ A_t\mathrm{id}; g\circ B_t\mathrm{id} \quad \text{(by the inductive hypothesis)}$$

$$=g\circ(p\rightarrow A_t\mathrm{id}; B_t\mathrm{id})=g\circ H_t\mathrm{id}$$

(f) $Hf=Pf\rightarrow Af; Bf$, where P, A, B are CLFs and $P_t=A_t=B_t$. By Proposition 2.4 we have $H_t g=P_t g=g\circ P_t\mathrm{id}$ (by the inductive hypothesis)$=g\circ H_t\mathrm{id}$. □

Appendix B. The combinator-form definition of the loop expression E_H

Given CLF H, we define as follows:

(a) if $Hf=f$ then $E_H=1$,

(b) if $Hf=(Gf)\circ a$ then $E_H=E_G\circ[1, a\circ 2]$,

(c) if $Hf=a\circ(Gf)$ then $E_H=a\circ E_G$,

(d) if $Hf=[g_1, g_2, \ldots, g_n]$ and for $1\leqslant i\leqslant n$,
(i) $g_i=a_i$ or $g_i=H_i f$, where H_i is a CLF,
(ii) if $g_i=H_i f$ then $H_{it}=H_t$,
then $E_H=[F_1, \ldots, F_n]$, where
$$F_i=\begin{cases} a_i\circ 2 & \text{if } g_i=a_i,\\ E_{Hi} & \text{if } g_i=H_i f,\end{cases}$$

(e) if $Hf=p\rightarrow Af; Bf$, where A, B are CLFs, then $E_H=p\circ 2\rightarrow E_A; E_B$,

(f) if $Hf=Pf\rightarrow Af; Bf$, where P, A and B are CLFs, s.t. $P_t=A_t=B_t$ then $E_H=E_p\rightarrow E_A; E_B$.

Appendix C. Proof of Proposition 2.6

The proof is again by structural induction. In the base case, $H=\mathrm{ID}$ and $Hf:x=f:x=E_H(f:x, x)=E_H(H_t f:x, x)$. Using the corresponding labelling of Definition 2.2, the inductive part of the proof falls into the five cases that follow. For each

case, in turn, we evaluate the proposed expression for $E_H(u, v)$ with $(H_t f):x$ and x substituted for u and v, respectively, and show that the result is equal to $Hf:x$.

(b) $E_H(H_t f:x, x) = E_G(H_t f:x, a:x)$

$= E_G((G_t f) \circ a:x, a:x)$ (by Proposition 2.3)

$= E_G(G_t f:y, y)$ (where $y = a:x$)

$= Gf:y$ (by the inductive hypothesis)

$= (Gf) \circ a:x = Hf:x.$

(c) $E_H(H_t f:x, x) = a:E_G(H_t f:x, x)$

$= a:E_G(G_t f:x, x)$ (by Proposition 2.3)

$= a:Gf:x$ (by the inductive hypothesis)

$= Hf:x.$

(d) $E_H(H_t f:x, x) = \langle F_1(H_t f:x, x), F_2(H_t f:x, x), \ldots, F_n(H_t f:x, x)\rangle$, where F_i $(1 \leqslant i \leqslant n)$ is defined as in Definition 2.5.
If $g_i = H_i f$ then $F_i(H_t f:x, x) = E_{H_i}(H_t f:x, x)$

$= E_{H_i}(H_{it} f:x, x)$ (by Proposition 2.3)

$= H_i f:x$ (by the inductive hypothesis)

If $g_i = a_i$ then $F_i(H_t f:x, x) = a_i:x$ and, so, $E_H(H_t f:x, x) = [g_1, g_2, \ldots, g_n]:x = Hf:x$.

(e) $E_H(H_t f:x, x) = \underline{\text{if}}\ p:x = T\ \underline{\text{then}}\ E_A(H_t f:x, x)\ \underline{\text{else}}\ E_B(H_t f:x, x)$.
Now by Proposition 2.3, for all functions, a, $H_t a = p \rightarrow A_t a; B_t a$. Thus,

$p:x = T \Rightarrow (H_t a):x = (A_t a):x$

$p:x = F \Rightarrow (H_t a):x = (B_t a):x$ and, so,

$E_H(H_t f:x, x) = \underline{\text{if}}\ p:x = T\ \underline{\text{then}}\ E_A(A_t f:x, x)\ \underline{\text{else}}\ E_B(B_t f:x, x)$

$= \underline{\text{if}}\ p:x = T\ \underline{\text{then}}\ Af:x\ \underline{\text{else}}\ Bf:x$ (by the inductive hypothesis)

$= (p \rightarrow Af; Bf):x = Hf:x.$

(f) $E_H(H_t f:x, x) = \underline{\text{if}}\ E_P(H_t f:x, x)\ \underline{\text{then}}\ E_A(H_t f:x, x)\ \underline{\text{else}}\ E_B(H_t f:x, x)$

$= \underline{\text{if}}\ E_P(P_t f:x, x)\ \underline{\text{then}}\ E_A(A_t f:x, x)\ \underline{\text{else}}\ E_B(B_t f:x, x)$
(by Proposition 2.3)

$= \underline{\text{if}}\ Pf:x\ \underline{\text{then}}\ Af:x\ \underline{\text{else}}\ Bf:x$ (by the inductive hypothesis)

$= Hf:x$ □

Appendix D. Outline of the method (b) of the proof of Theorem 3

We will use the following notation:

$$\text{list} = p \circ \text{hd} \to \text{id}; L\text{list},$$

$$f' = \text{null} \circ 2 \to 1; Ff',$$

$$f'' = p \to q; k,$$

where $k = [q \circ \text{hd}, \text{tl}] \circ \text{list} \circ [\text{id}]$, and for functions a,

$$La = L_t a = a \circ a\text{l} \circ [H_t\text{id} \circ \text{hd}, \text{id}],$$

$$Fa = F_t a = a \circ [E_H \circ [1, \text{hd} \circ 2], \text{tl} \circ 2].$$

Also, let $k_i = [q \circ \text{hd}, \text{tl}] \circ L^i\text{id} \circ [\text{id}]$ $(i \geqslant 0)$.

The following nine intermediate results are all easy to prove.

(i) $H_t^i p = L_t^i(p \circ \text{hd}) \circ [\text{id}]$ for $i > 0$
since $L_t^i(p \circ \text{hd}) = p \circ \text{hd} \circ (a\text{l} \circ [H_i\text{id} \circ \text{hd}, \text{id}])^i$

$$= p \circ H_t\text{id} \circ \text{hd} \circ (a\text{l} \circ [H_t\text{id} \circ \text{hd}, \text{id}])^{i-1}$$

$$= p \circ (H_t\text{id})^i \circ \text{hd},$$

and using Proposition 2.4.

(ii) $H_t^i p \Rightarrow \text{list} \circ [\text{id}] = L^i\text{id} \circ [\text{id}]$ and $H_t^i p \Rightarrow k = k_i$
by the LET applied to the function list, and since if $a \Rightarrow b = c$, then $a \circ d \Rightarrow b \circ d = c \circ d$ for all functions a, b, c, d.

(iii) $\text{tl}^j \circ L^i\text{id} = L^{i-j}\text{id}$ for $0 \leqslant j \leqslant i$

(iv) $F_t^i(\text{null} \circ 2) = \text{null} \circ \text{tl}^i \circ 2$
since $F_t^i(\text{null} \circ 2) = \text{null} \circ 2 \circ [E_H \circ [1, \text{hd} \circ 2], \text{tl} \circ 2]^i$

$$= \text{null} \circ \text{tl} \circ 2 \circ [E_H \circ [1, \text{hd} \circ 2], \text{tl} \circ 2]^{i-1}$$

(v) $H_t^i p \Rightarrow F_t^i(\text{null} \circ 2) \circ k_i = \boldsymbol{T}$
using (i), (iii) and since if $a \Rightarrow b = c$, then $a \Rightarrow d \circ b = d \circ c$ for all functions a, b, c, d.

(vi) $F^j 1 \circ k_i = E_H \circ [F^{j-1} 1 \circ k_i, (H_t\text{id})^{i-j}]$ for $0 \leqslant j \leqslant i$ similarly.

(vii) $F^i 1 \circ k_i \circ H_t\text{id} = F^i 1 \circ k_{i+1}$.
In fact, we show that $F^j 1 \circ k_i \circ H_t\text{id} = F^j 1 \circ k_{i+1}$ for $0 \leqslant j \leqslant i$ by induction on j. Since $k_0 = [q, [\,]]$ and $k_1 = [q \circ H_t\text{id}, [\text{id}]]$ the result is true for $j = 0$. Assume it to be true for $j = J - 1 \geqslant 0$. Then by (vi),

$$F^J 1 \circ k_{i+1} = E_H \circ [F^{J-1} 1 \circ k_{i+1}, (H_t\text{id})^{i+1-J}] \qquad \text{for } i+1 \geqslant J$$

$$= E_H \circ [F^{J-1} 1 \circ k_i, (H_t\text{id})^{i-J}] \circ H_t\text{id} \qquad \text{for } i \geqslant J$$
(by the inductive hypothesis)

$$= F^J 1 \circ k_i \circ H_t\text{id} \quad \text{(by (vi))}.$$

(viii) $H_t^i p \Rightarrow F^i 1 \circ k_i = H^i q$.
This also follows by induction using the facts that
$H^i q = E_H \circ [H^{i-1} q \circ H_t \mathrm{id}, \mathrm{id}]$ (by Proposition 2.4)
and $F^i 1 \circ k_i = E_H \circ [F^{i-1} 1 \circ k_{i-1} \circ H_t \mathrm{id}, \mathrm{id}]$ (by (vi) and (vii))
(ix) $H_t^i p \Rightarrow f = f''$ for all $i \geqslant 0$.

References

[1] J.W. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21**(8) (1978) 613–641.

[2] J.W. Backus, The algebra of functional programs: function level reasoning, linear equations and extended definitions, in: *Formalization of Programming Concepts*, Lecture Notes in Computer Science, Vol. 107 (Springer, New York, 1981) 1–43.

[3] J.W. Backus, From function level semantics to program transformation and optimisation, in: *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science, Vol. 185 (Springer, New York, 1985).

[4] F.L. Bauer and H. Wossner, *Algorithmic Language and Program Development* (Springer, Berlin, 1982).

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

[6] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *J. ACM* **24**(1) (1977).

[7] R.M. Burstall, D.B. McQueen and D.T. Sannella, HOPE: An experimental applicative language, in: *Proc. 1st Internat. LISP Conf.*, Stanford (1980) 136–143.

[8] A.K. Chandra, Efficient compilation of linear recursive programs, in: *14th Ann. IEEE Symp. on Switching and Automata Theory*, Iowa City, IA (1973) 16–25.

[9] E.W. Dijkstra, Lecture Notes on Program Development, in: *Proc. Internat. Summer School on Program Construction*, Marktoberdorf, West Germany (1978).

[10] J.S. Givler and R.B. Kieburtz, Schema recognition for program transformation, in: *ACM Symp. on LISP and Functional Programming* (1984) 74–85.

[11] P.G. Harrison, Linearisation: An Optimisation for Non-linear Functional Programs, *Sci. Comput. Programming* **10** (1988) 281–318.

[12] R.B. Kieburtz and J. Shultis, Transformations of FP Program Schemes, in: *Proc. ACM Conf. on Functional Languages and Computer Architecture*, Portsmouth, NH (1981).

[13] D.E. Knuth and P. Bendix, Simple word problems in universal algebras. Computational Problems in Abstract Algebra (Pergamon Press, Oxford, 1970) 263–297.

[14] Z. Manna and R. Waldinger, The origin of the binary-search paradigm, SRI International, Technical Note 352, 1985.