The Mechanical Transformation of Data Types

P. G. HARRISON* AND H. KHOSHNEVISAN
Department of Computing, Imperial College of Science and Technology, University of London, London SW7 2BZ

The efficient implementation of abstract data types and all functions that manipulate them would permit application-oriented solutions to be developed without having to take undue account of executional properties. The synthesis of efficient concrete types and functions forms the basis of the present paper, which appeals to a theory of inverse functions for additional axioms to augment those of a first-order functional algebra. These axioms are then applied in the simplification of the combinator-expressions arising in the synthesis of the functions between the concrete types. We also show how the abstraction function itself may be deduced in certain situations where it is required to optimise particular operations on an abstract type. In addition to possessing rigorous mathematical foundations, the function-level axioms are more generally applicable than previous approaches to this problem, and induce a more mechanisable rewrite-based transformation system.

Received November 1987, revised December 1990

1. INTRODUCTION

The requirements that software be both correct and efficient often conflict, so impeding its design and maintenance. Using the transformational approach, programs are developed by concentrating only on the correctness, clarity, reliability and maintenance aspects, and a series of meaning-preserving transformations are then applied to the initial specification to produce an efficient implementation. In this way the problem is initially programmed at the appropriate abstract level, by choosing and specifying appropriate abstract data types in particular. In a transformation-based program development methodology, consideration of efficient representations of these data types is delayed until implementation, where the operations required on a data type will be implemented using the structures that are available in the chosen implementation language. For example, priority queues can be efficiently implemented as binomial trees,[13] or trees can be efficiently implemented as vectors, as Floyd shows in Ref. 7. Alternatively, one may be seeking an implementation of a data type such that a particular set of operations defined on it are optimised in some way; after all, the purpose of any representation is to facilitate some kind of computation. For example, having used a high-level representation of the data type 'set of numbers', we may wish to find an implementation that optimises the operations *minimum* or *maximum* on it. To optimise the operation *minimum*, one possible representation of such a set might be an ordered list of numbers – in ascending order. In this way the operation *minimum* would simply be implemented as 'head'. Similarly, if instead we required *maximum* to be optimised, the list would be organised in descending order. Of course, as Hoare points out in Ref. 10, one should verify the correctness of the implementation of the operations defined on a data type, and this is usually done by specifying a *representation* function, which specifies how the concrete data type represents the abstract one. The task of verifying the implementation then becomes that of showing that, under the representation function, every operation implemented performs its intended purpose, as defined by the axioms given for the abstract type.

* To whom correspondence should be addressed.

Since program transformation is meaning-preserving, its use in the synthesis of a data type's implementation also provides the necessary verification at the same time. But in addition to separating the concerns for correctness and efficiency, the transformational approach brings with it the possibility of mechanisation of the crucial step of designing the structures to represent abstract data types efficiently. A popular method for specifying a data type defines the set of all primitive operations on it, and is known as the algebraic or equational method.[4,8,12] Most of the work done to date on the synthesis of data type implementations has been based on the more traditional styles of program transformation; for example, Darlington[5,6] describes how this can be achieved by applying the program transformation methodology of Burstall and Darlington[3] to the type's defining equations and to the function mapping the concrete type to the abstract one.

The majority of contemporary program transformation schemes require a relatively large amount of ingenuity – 'eureka-steps' – because they analyse expressions written in lambda-calculus-based functional languages. In these languages a function's definition comprises equations giving the results of applications of that function in terms of the particular arguments to which it is applied. The resulting transformations therefore rely on an analysis not only of functions but also of *objects* – functions' arguments and results. However, by compiling such programs into a combinatory, i.e. variable-free, representation, it is possible for the analysis to achieve more powerful, automatic, and generally applicable transformations. This is due to the elimination of the concern for the domain of objects, since function definitions are now expressed purely in terms of other functions.

The basis of the data-type transformations developed in this paper is a 'commutative square' of functions comprising horizontally a user-defined function and its corresponding concrete version, and vertically the abstraction functions between the domain- and range-types. Then concrete functions are obtained as compositions involving abstraction functions, their inverses and user-defined functions. This is explained in Section 2, after which the main results of our analysis of inverse functions,[9] are summarised and combined with Backus's algebraic laws,[1,2] to yield a set of rewrite rules which we use to drive our transformation system. In Section 5 we present a scheme whereby the abstraction function may also be deduced in certain situations where it is required to optimise given operations on an abstract type as discussed above, and illustrate the technique with examples. The main results of the paper are summarised in Section 6, along with some open questions and suggestions for future applications and research.

Definition of symbols

We use the following notation throughout the paper, much of which is in the FP style of Ref. 1.

T, F	truth values 'TRUE' and 'FALSE'
$\circ$	function composition, i.e. $(f \circ g)x = f(gx)$ for all objects x
$[,]$	function construction, i.e. $[f,g]x = \langle fx, gx\rangle$ for all objects x
$f = p \to q; r$	$fx =$ if $px = T$ then qx else if $px = F$ then rx else $\perp$ for all objects x
a	constant function, i.e. $\mathbf{a}x = a$ for all objects $x \neq \perp$, $\mathbf{a}\perp = \perp$ (e.g. **0**, **T**, **1** etc.)
$\perp$	the undefined object, 'bottom'
id	the identity function, i.e. $\mathrm{id}\,x = x$ for all objects x.
hd, tl, null, add1, sub1, cons, eq, etc.	Standard primitive functions

2. MAPPINGS ON DATA TYPES

2.1 General approach

Consider the pair of abstract data types α, β, and the corresponding pair α', β', which we take to be concrete types that provide realisations of α, β respectively. Then, given the function $f: \alpha \to \beta$, it may be required to synthesise a corresponding function, say $f': \alpha' \to \beta'$, which performs operations on objects of type α' which are isomorphic in some sense to the operations performed by f on corresponding objects of type α. The function f' might then be the 'implementation function' of f, which can be executed more efficiently than f and, together with the types α' and β', removes from the run-time system the need to represent the abstract types at all. Let the abstraction function for abstract type σ be $\mathrm{abs}_\sigma: \sigma' \to \sigma$ where σ' is the concrete, implementation type corresponding to σ. In this paper we assume abstraction functions are single-valued. However, as we will note in later sections, a many-valued function abs_σ could provide the basis for representing sets (of type σ) by lists (of type σ') which do not contain duplicate elements.

The concrete function f' must form a commutative square in the following diagram.

$$\begin{tikzcd}
\alpha \arrow[r, "f"] & \beta \\
\alpha' \arrow[u, "\mathrm{abs}_\alpha"] \arrow[r, "f'"'] & \beta' \arrow[u, "\mathrm{abs}_\beta"']
\end{tikzcd}$$

In other words, $f \circ \mathrm{abs}_\alpha = \mathrm{abs}_\beta \circ f'$, or $f' = \mathrm{abs}_\beta^{-1} \circ f \circ \mathrm{abs}_\alpha$, assuming that the inverse-function, abs_β^{-1}, exists. Thus, for example, if $\alpha = \beta$ we have, dropping the subscript from abs_α where there is no ambiquity,

$$\begin{tikzcd}
\alpha \arrow[r, "f"] & \alpha \arrow[d, "\mathrm{abs}^{-1}"] \\
\alpha' \arrow[u, "\mathrm{abs}"] \arrow[r, "f'"'] & \alpha'
\end{tikzcd}$$

so that $f' = \mathrm{abs}^{-1} \circ f \circ \mathrm{abs}$. For $\beta = \beta'$, we have a triangle:

$$\begin{tikzcd}
\alpha \arrow[r, "f"] & \beta \\
\alpha' \arrow[u, "\mathrm{abs}"] \arrow[ur, "f'"'] &
\end{tikzcd}$$

so that $f' = f \circ \mathrm{abs}$. Equivalently, we may simply choose $\mathrm{abs}_\beta = \mathrm{id}$, the identity function. We take the fixed functions to be polymorphic where possible, so that the identity and selector functions, for example, apply to all types.

For a product type of the form $\gamma = \alpha \times \beta$, we define

$$\mathrm{abs}_\gamma: \alpha' \times \beta' \to \alpha \times \beta = [\mathrm{abs}_\alpha \circ 1, \mathrm{abs}_\beta \circ 2]$$

and in general if $\sigma = \sigma_1 \times \ldots \times \sigma_n$,

$$\mathrm{abs}_\sigma = [\mathrm{abs}_{\sigma 1} \circ 1, \ldots, \mathrm{abs}_{\sigma n} \circ n]$$

and

$$\mathrm{abs}_\sigma^{-1} = [\mathrm{abs}_{\sigma 1}^{-1} \circ 1, \ldots, \mathrm{abs}_{\sigma n}^{-1} \circ n].$$

(Note that this yields $\mathrm{abs}_\sigma \circ \mathrm{abs}_\sigma^{-1} = \mathrm{id}$ as required; in the next section we will define abs_σ^{-1} so that $\mathrm{abs}_\sigma^{-1} \circ \mathrm{abs}_\sigma = \mathrm{id}$ also.)

Thus, for example, the list constructor function, cons, produces the following square

$$\begin{tikzcd}
\beta \times \alpha \arrow[r, "\mathrm{cons}"] & \alpha \arrow[d, "\mathrm{abs}_\alpha^{-1}"] \\
\beta \times \alpha' \arrow[u, "\mathrm{abs}_\alpha"] \arrow[r, "\mathrm{cons}'"'] & \alpha'
\end{tikzcd}$$

where $\mathrm{abs} = [1, \mathrm{abs}_\alpha \circ 2]$, α' representing the concrete form of lists, α, with data objects of type β.

2.2 The concrete versions of user-defined functions

In general, given the abstraction functions for types α and β and the abstract function $f: \alpha \to \beta$, we would like to generate automatically the concrete version of f, i.e. the function f' that completes the appropriate commutative square, as an implementation of f. In other words, we wish to find $f' = \mathrm{abs}_\beta^{-1} \circ f \circ \mathrm{abs}_\alpha$, expressed purely in terms of the types α', β', so that no representation of the abstract type will be required at run-time.

Consider now first-order functional expressions E with the syntax

$$E ::= A \mid E1 \circ E2 \mid [E1, \ldots, En] \mid E1 \to E2; E3$$

where A denotes the primitive functions. For primitive function, a of type $\alpha \rightarrow \beta$, the concrete version is $a' = \text{abs}_\beta^{-1} \circ a \circ \text{abs}_\alpha$, which is synthesised using the axioms of the types α, α', β, β' and the definition of a. Note that for a polymorphic function, p, we have $p' = p$, e.g. $1' = 1$. Otherwise it is easy to see, by induction on the type structure of E, that E' is that expression obtained by replacing in E each occurrence of every primitive function, a, by its concrete version, a'.

To prove this claim, let $E_{\alpha\beta}$ denote an expression, E, of type $\alpha \rightarrow \beta$, and suppose first that $E_{\alpha\gamma} = E1_{\beta\gamma} \circ E2_{\alpha\beta}$. Then

$$\begin{aligned} E' &= \text{abs}_\gamma^{-1} \circ E1_{\beta\gamma} \circ E2_{\alpha\beta} \circ \text{abs}_\alpha \\ &= \text{abs}_\gamma^{-1} \circ E1_{\beta\gamma} \circ \text{abs}_\beta \circ \text{abs}_\beta^{-1} \circ E2_{\alpha\beta} \circ \text{abs}_\alpha \\ &\quad \text{(assuming abs}_\beta \text{ is single-valued)} \\ &= E1' \circ E2'. \end{aligned}$$

Now, suppose that $E_{\alpha\gamma} = [E1_{\alpha\gamma_1}, \ldots, En_{\alpha\gamma_n}]$, where $\gamma = \gamma_1 \times \ldots \times \gamma_n$. Then

$$\begin{aligned} E' &= \text{abs}_\gamma^{-1} \circ [E1_{\alpha\gamma_1}, \ldots, En_{\alpha\gamma_n}] \circ \text{abs}_\alpha \\ &= [\text{abs}_{\gamma_1}^{-1} \circ E1_{\alpha\gamma_1} \circ \text{abs}_\alpha, \ldots, \text{abs}_{\gamma_n}^{-1} \circ En_{\alpha\gamma_n} \circ \text{abs}_\alpha] \\ &= [E1'_{\alpha\gamma_1}, \ldots, En'_{\alpha\gamma_n}]. \end{aligned}$$

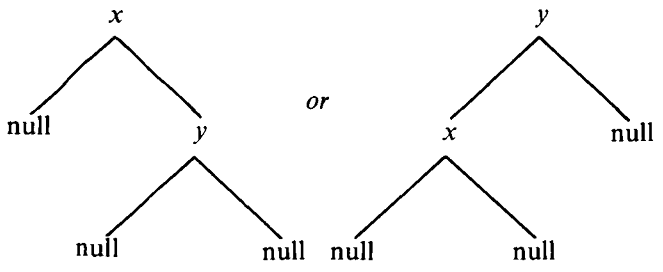
If $E_{\alpha\gamma} = E1_{\alpha B} \rightarrow E2_{\alpha\gamma}; E2_{\alpha\gamma}$ where B denotes the boolean type comprising the truth values T and F and the value 'undefined', the inductive step follows simply, since $B \equiv B'$ and

$$E' = \text{id} \circ E1_{\alpha B} \circ \text{abs}_\alpha \rightarrow \text{abs}_\gamma^{-1} \circ E2_{\alpha\gamma} \circ \text{abs}_\alpha; \quad \text{abs}_\gamma^{-1} \circ E2_{\alpha\gamma} \circ \text{abs}_\alpha.$$

Finally, suppose that f is a first-order, recursively defined function. Then its defining expression has the syntax given above for E, extended to admit the function variable f as an additional syntactic type. Then the concrete version, f', of f has for its defining expression the defining expression of f with all occurrences of primitive functions replaced by their concrete versions, and f replaced by f'. This follows by precisely the same induction as that used above, the base case of $E = f$ being trivial. Strictly, this follows since the corresponding ascending Kleene chains have the same limit since for all n $(E^n \bot)' = (E')^n \bot$. This limit is the least fixed point of the equation for f'.

2.3 Non-uniqueness of concretisation functions

Although the abstraction relations, abs_σ, are assumed to be single-valued, being so defined by the programmer (or perhaps automatically synthesised as discussed in Section 5), they are not in general 1–1. This is not surprising, since there will typically be a number of different objects in the domain of the concrete type σ' that correspond to any given object in the abstract domain of σ. For example, a list may be realised by a vector–integer pair, in which the integer is the length of the list, $n \geqslant 0$ say, and the ith element of the list is the same as the $(n-i+1)$th component of the vector. Thus the list elements are stored in reverse order in the vector, and the integer n is the 'head-pointer'; this facilitates an efficient 'cons' operation. There are therefore an infinite number of vectors corresponding to any given list: all components after the nth in a vector of dimension greater than n representing a list of length n are arbitrary. Another example is the realisation of a list by a binary tree which is defined to be either empty or else a triple comprising a root-node and two binary trees (the left and right sub-trees). The list represented by the binary tree b is then formed by appending three lists: the list corresponding to the left sub-tree of b, the singleton-list comprising the element at the root-node of b, the list corresponding to the right sub-tree of b. For a list of length n, there are then n sets of such possible binary trees, each tree in a given set having a different element at its root-node. For example, if $n = 2$, there are the two possible binary trees:



Formal definitions of these two implementations of lists are given in Section 4, along with the mechanical synthesis of the corresponding data-type transformations and of certain user-defined functions on lists.

However, in practice there will be only one object of a concrete type, σ', which is used to represent a given object in the abstract type σ. For example, a list of length n would typically be represented by a unique vector with n components (together with its dimension, n) or by a unique, evenly balanced tree. The abstraction function therefore has a unique inverse function, which will not, in general, be onto, and we define, for abstract type σ, the single-valued concretisation relation, $\text{conc}_\sigma = \text{abs}_\sigma^{-1}$. Here, then, we are using the normal notion of a function and we assume that any abstraction function, abs_σ, has a unique inverse, conc_σ; the more general approach is suggested in Ref. 9.

Our transformation system operates by successively rewriting the expressions defining the functions conc and f' defined above, in terms of abs and f, using a set of axioms which is described in Section 3. This set includes the FP axioms,[1] and those relating to function-inversion,[9] which are generally applicable. In addition, the user-supplied definitions of the abstract data types are also used as the basis of further axioms for their particular transformations. In this way, many functions $f' : \alpha' \rightarrow \beta'$ corresponding to $f : \alpha \rightarrow \beta$ supplied by the programmer may be synthesised. Several examples of the applications of this type of transformation are given in Sections 4 and 5, in the latter of which the function conc is also synthesised.

3. THE TRANSFORMATION SYSTEM: AXIOMS AND RULES

3.1 A functional language for transformation

In order to define transformations on data types of the sort considered in the previous section, we need a language in which we can express the inverse of every function (to obtain conc) and the composition of functions. We consider first-order functions and give all functional expressions in variable-free form. Thus a language based on first-order categorical combinatory logic augmented to facilitate the definition of inverse functions is suitable, certainly providing the required composition operator. We therefore adopt a suitably augmented variant of an FP-like language.[1] All the primitive functions of FP are assumed to be available, together with their corresponding inverses, but the selector primitives of FP, 1, 2, ... are represented by n_m, where $m \geqslant n \geqslant 1$. The primitive n_m reads as 'select the nth element of a sequence of m elements'. Similar dialects of FP have been used in other transformation systems, e.g. Kieburtz and Shultis,[11] since these selectors convey some typing information, namely the size of the sequences to which they can be applied.

We also introduce logical function variables which represent (unknown) functions, and appear in the function-level expressions for the inverses of selector functions and of constructions of functions. Logical function variables are denoted by the infinite set of primitives ?1, ?2, ... and are bound to ground function values by function-level unification, which we introduce below. Examples of inverse functions with occurrences of logical variables are:

$$\mathrm{hd}^{-1} \circ f = [f, ?1, \ldots] = \mathrm{cons} \circ [f, ?1]$$

$$\mathrm{tl}^{-1} \circ f = \mathrm{cons} \circ [?1, f]$$

$$5_9^{-1} \circ f = [?1, ?2, ?3, ?4, f, ?5, \ldots, ?8]$$

In general, the program-forming operations of FP, abbreviated to PFOs, must be generalised to operate on power domains so that they can be applied to set-valued functions and so accommodate inverse functions which are not single-valued. Although the definition of composition need not be changed, the generalisations needed for the construction and conditional PFOs are non-trivial. For example, the extended construction-PFO is defined by

$$[f_1, \ldots, f_n]X = \{<y_1, \ldots, y_n> \mid y_i \in f_i\{x\}^*,\ 1 \leqslant i \leqslant n, x \in X\}$$

for functions $f_1, \ldots, f_n$ and set X of objects, where $\{x\}^*$ denotes the Scott-closure of $\{x\}$ in the power domain of which X is a member.

The conditional-PFO becomes even more complex. However, for our purposes, all functions may be assumed single-valued, and the PFOs need not be so extended.

We also define the PFO 'unity-at-function-level', denoted by $\lll\ \ggg$, which is used for the inversion of constructions of functions. $\lll\ \ggg$ takes two or more functions of the same arbitrary type as arguments and returns a new function of that type. The definition of the $\lll\ \ggg$-PFO is given by axioms in the next section as first-order unification, function-representations being the objects of our unification. Since unification is associative and commutative, $\lll\ \ggg$ also has these properties.

3.2 Axioms

In the general case of inverse-function synthesis, we must introduce set-valued functions and so generalise some of the definitions of the PFOs as noted in the previous section. As a result, some of the FP axioms of Backus[1] then do not hold as they stand. For example, it may no longer be the case that $[f, g] \circ h = [f \circ h, g \circ h]$ for all functions f, g, h, although it does remain the case that $n_m \circ [f_1, \ldots, f_m] = f_n$ for $1 \leqslant n \leqslant m$. However, for our purposes, where all functions are single-valued, we may include all of the FP axioms in our system, and we also introduce axioms which involve the extensions that we have made, most notably the PFO $\lll\ \ggg$ and the inverses of primitive functions.

For any functional expression f, we define the axioms:

$\lll ?1, f \ggg = f$ (including $\lll f \ggg = f$) (and thus by associativity and commutativity $\lll ?1, \ldots, ?i, f, ?(i+1), \ldots, ?k \ggg = f$ for $k \geqslant i \geqslant 1$). (A 1)

For the inverse selector function i_j^{-1} $(0 \leqslant i \leqslant j)$

$$i_j^{-1} \circ f = [?1, \ldots, ?(i-1), f, ?(i+1), \ldots, ?j] \quad \text{(A 2)}$$

$$\lll [a1_1, \ldots, a1_n], [a2_1, \ldots, a2_n], \ldots, [am_1, \ldots, am_n] \ggg = [\lll a1_1, a2_1, \ldots, am_1 \ggg, \lll a1_2, a2_2, \ldots, am_2 \ggg, \ldots, \lll a1_n, a2_n, \ldots, am_n \ggg] \quad (m, n \geqslant 1) \quad \text{(A 3)}$$

$$\mathrm{hd}^{-1} \circ f = \mathrm{cons} \circ [f, ?1] \quad \text{(A 4)}$$

$$\mathrm{tl}^{-1} \circ f = \mathrm{cons} \circ [?1, f] \quad \text{(A 5)}$$

$$\mathrm{cons}^{-1} = [\mathrm{hd}, \mathrm{tl}] \quad \text{(A 6)}$$

$h \circ \lll f, g \ggg = \lll h \circ f, h \circ g \ggg$
and
$\lll f, g \ggg \circ h = \lll f \circ h, g \circ h \ggg$
for all functions f, g, h (A 7)

$\lll f \circ h, g \circ k \ggg = \lll f \circ 1_2, g \circ 2_2 \ggg \circ [h, k]$ for all functions f, g, h (A 8)

$$\lll p \to q; r, p' \to q'; r' \ggg = p \to (p' \to \lll q, q' \ggg; \lll q, r' \ggg); (p' \to \lll r, q' \ggg; \lll r, r' \ggg) \quad \text{(A 9}a\text{)}$$

if p implies p'
$$\lll p \to q; r, p' \to q'; r' \ggg = p \to \lll q, q' \ggg; (p' \to \lll r, q' \ggg; \lll r, r' \ggg) \quad \text{(A 9}b\text{)}$$

Note that these axioms are consequences of object-level unification semantic properties, abstracted to the function level (i.e. the variables have been abstracted).

Two 'theorems' that derive from these axioms, and which we will find important for removing unification from the definitions of inverses, are

$$\lll 1_n^{-1} \circ a_1, \ldots, n_n^{-1} \circ a_n \ggg = [a_1, \ldots, a_n] \quad \text{(T 1)}$$

which follows by repeated application of (A 2), and then (A 3) and (A 1)

$$\lll \mathrm{hd}^{-1} \circ f, \mathrm{tl}^{-1} \circ g \ggg = \mathrm{cons} \circ [f, g] \quad \text{(T 2)}$$

which follows by application of (A 4), (A 5), (A 7) and then (A 3).

3.3 Function inversion

We now summarise the main results for constructing the inverses of various fixed and recursively defined functions. First, the inverse of a composition of two functions is given by

$$(f \circ g)^{-1} = g^{-1} \circ f^{-1} \quad \text{(I 1)}$$

and the inverse of the construction of n functions is given by

$$[f_1, \ldots, f_n]^{-1} = \lll f_1^{-1} \circ 1_n, \ldots, f_n^{-1} \circ n_n \ggg \quad \text{(I 2)}$$

The proofs of these results can be found in Ref. 9, where a similar result for conditionals is also derived.

Now consider the function f defined recursively by the equation $f = p \to q; E$ for fixed functions p, q and expression E having the following syntax

$$E ::= f \mid a \mid E \circ E \mid [E, E, \ldots, E],$$

where the syntactic type a denotes fixed functions and f is the function variable. (The further syntactic type, $a \to E; E$ may also be added if we include the result for the inverse of a conditional. However, we will not find this necessary here.) We may now write down an expression for E^{-1} by hierarchical decomposition into the syntactic types corresponding to composition and construction.

For example, if $E = E_1 \circ E_2$, then $E^{-1} = E_2^{-1} \circ E_1^{-1}$. If $E_1 = [E_3, E_4]$ and $E_2 = \text{add1}$, then $E_1^{-1} = \langle\!\langle E_3^{-1} \circ 1_2, E_4^{-1} \circ 2_2 \rangle\!\rangle$ and $E_2^{-1} = \text{sub1}$ (a primitive type) and so on. Moreover, again by a case analysis of the syntactic types, we can see that for any such expression E, E^{-1} may be written $H'f^{-1}$ for some fixed functional H'. The rigorous proofs are very easy inductions on the types. Given E with this syntax, a functional H defined by $Hf = E$ is called *composite*.

The main result of the inversion of recursively defined functions is the following, the proof again appearing in Ref. 9.

Let f be defined by $f = p \to q; Hf$ for composite functional H and continuous functions p, q. Assuming q is 1–1,

$$f^{-1} = p \circ q^{-1} \to q^{-1}; H'f^{-1}, \qquad \text{(I 3)}$$

where H' is defined by $H'f^{-1} = (Hf)^{-1}$ for function variable f. (In fact f^{-1} is the greatest fixed point of this equation.)

4. TRANSFORMATION FOR AUTOMATIC IMPLEMENTATION OF DATA TYPES

4.1 Representing lists as trees

We might wish to represent a list as a tree, for example to take advantage of parallel evaluation. Assume that we are given the following data definition:

data tree α == empty ++ node (tree $\alpha \times \alpha \times$ tree α)

together with the abstraction function, abs, which shows how the concrete data type, tree, maps to the abstract type, list, i.e.

```
dec  abs : tree α → list α
     abs (empty) = nil
     abs (node (t1, n, t2)) = abs (t1) < > n :: abs (t2)
```

We would like to synthesise the concrete versions of the abstract functions that operate on lists, and in particular, we synthesise below a function revtree corresponding to the function rev that reverses lists.

The function abs can be expressed in variable-free form as follows,

$$\text{abs} = \text{isempty} \to \mathbf{nil}; \quad \text{append} \circ [\text{abs} \circ 1_3, \text{cons} \circ [2_3, \text{abs} \circ 3_3]]$$

Now by our inversion results we can synthesise the inverse function of abs, i.e. conc, as follows.

conc = abs^{-1}

$= \text{null} \to \text{empty}; (\text{append} \circ [\text{abs} \circ 1_3, \text{cons} \circ [2_3, \text{abs} \circ 3_3]])^{-1}$ by (I 3)

$= \text{null} \to \text{empty}; [\text{abs} \circ 1_3, \text{cons} \circ [2_3, \text{abs} \circ 3_3]]^{-1} \circ \text{append}^{-1}$ by (I 1)

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{abs}^{-1} \circ 1_2, (\text{cons} \circ [2_3, \text{abs} \circ 3_3])^{-1} \circ 2_2 \rangle\!\rangle \circ \text{append}^{-1}$ by (I 2)

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{abs}^{-1} \circ 1_2, [2_3, \text{abs} \circ 3_3]^{-1} \circ \text{cons}^{-1} \circ 2_2 \rangle\!\rangle \circ \text{append}^{-1}$ by (I 1)

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{abs}^{-1} \circ 1_2, \langle\!\langle 2_3^{-1} \circ 1_2, 3_3^{-1} \circ \text{abs}^{-1} \circ 2_2 \rangle\!\rangle \circ \text{cons}^{-1} \circ 2_2 \rangle\!\rangle \circ \text{append}^{-1}$ by (I 2)

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{conc} \circ 1_2, \langle\!\langle 2_3^{-1} \circ 1_2, 3_3^{-1} \circ \text{conc} \circ 2_2 \rangle\!\rangle \circ \text{cons}^{-1} \circ 2_2 \rangle\!\rangle \circ \text{append}^{-1}$

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{conc} \circ 1_2, \langle\!\langle 2_3^{-1} \circ 1_2, 3_3^{-1} \circ \text{conc} \circ 2_2 \rangle\!\rangle \circ [\text{hd}, \text{tl}] \circ 2_2 \rangle\!\rangle \circ \text{append}^{-1}$, by (A 6)

$= \text{null} \to \text{empty}; \langle\!\langle 1_3^{-1} \circ \text{conc} \circ 1_2, \langle\!\langle 2_3^{-1} \circ \text{hd} \circ 2_2, 3_3^{-1} \circ \text{conc} \circ \text{tl} \circ 2_2 \rangle\!\rangle \rangle\!\rangle \circ \text{append}^{-1}$ by (A 7)

$= \text{null} \to \text{empty}; \langle\!\langle [\text{conc} \circ 1_2, ?1, ?2], [?3, \text{hd} \circ 2_2, \text{conc} \circ \text{tl} \circ 2_2] \rangle\!\rangle \circ \text{append}^{-1}$

$= \text{null} \to \text{empty}; [\text{conc} \circ 1_2, \text{hd} \circ 2_2, \text{conc} \circ \text{tl} \circ 2_2] \circ \text{append}^{-1}$

This systematic generation of conc is clearly correct, but the shape of the concrete tree depends on how append^{-1} (the function 'split' that splits a list into two lists) is defined. Although, in general, the function split is many-valued, we assume here that append^{-1} is single-valued, some canonical splitting of its argument being chosen for the concretisation, as discussed in Section 2. In practice, a balanced-tree representation would normally be required to enhance the exploitation of parallelism. For example, the pair of lists returned by an application of append^{-1} would be such that their lengths were equal or the second had one more element than the first. In fact we need not assume append^{-1} is single-valued, whereupon conc too will be many-valued. In this way we would derive a recursive definition for revtree which is an approximation (under the ordering of our function space). The fixed point for revtree that we will define shortly certainly satisfies this relation but is not the only solution.

Now since three of the functions in our commutative square are defined we can proceed to synthesise revtree. We will make use of the following additional axiom which is actually an alternative definition of the function rev:

$$\text{rev}(x <> y :: z) = \text{rev}(z) <> y :: \text{rev}(x)$$

i.e. $\text{rev} \circ \text{append} \circ [a, \text{cons} \circ [b, c]] = \text{append} \circ [\text{rev} \circ c, \text{cons} \circ [b, \text{rev} \circ a]]$

where a, b and c are any arbitrary functional expressions.

Now from the commutative diagram, omitting the subscripts from the selector functions for brevity, we have

revtree = conc ∘ rev ∘ abs

$= \text{isempty} \to \text{conc} \circ \text{rev} \circ \mathbf{nil}; \text{conc} \circ \text{rev} \circ \text{append} \circ [\text{abs} \circ 1, \text{cons} \circ [2, \text{abs} \circ 3]]$

by substituting for abs

$= \text{isempty} \to \text{conc} \circ \mathbf{nil}; \text{conc} \circ \text{rev} \circ \text{append} \circ [\text{abs} \circ 1, \text{cons} \circ [2, \text{abs} \circ 3]]$

from the definition of rev

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; \mathrm{conc} \circ \mathrm{rev} \circ \mathrm{append} \circ [\mathrm{abs} \circ 1, \mathrm{cons} \circ [2, \mathrm{abs} \circ 3]]$$

by definition of conc

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; \mathrm{conc} \circ \mathrm{append} \circ [\mathrm{rev} \circ \mathrm{abs} \circ 3, \mathrm{cons} \circ [2, \mathrm{rev} \circ \mathrm{abs} \circ 1]]$$

by applying the additional axiom

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; (\mathrm{null} \rightarrow \mathrm{empty}; [\mathrm{conc} \circ 1_2, \mathrm{hd} \circ 2_2, \mathrm{conc} \circ \mathrm{tl} \circ 2_2] \circ \mathrm{append}^{-1}) \circ \mathrm{append} \circ [\mathrm{rev} \circ \mathrm{abs} \circ 3, \mathrm{cons} \circ [2, \mathrm{rev} \circ \mathrm{abs} \circ 1]]$$

by substituting for conc

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; [\mathrm{conc} \circ 1_2, \mathrm{hd} \circ 2_2, \mathrm{conc} \circ \mathrm{tl} \circ 2_2] \circ \mathrm{append}^{-1} \circ \mathrm{append} \circ [\mathrm{rev} \circ \mathrm{abs} \circ 3, \mathrm{cons} \circ [2, \mathrm{rev} \circ \mathrm{abs} \circ 1]]$$

since null ∘ append ∘ [rev ∘ abs ∘ 3, cons ∘ [2, rev ∘ abs ∘ 1]] can never be true due to the use of cons

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; [\mathrm{conc} \circ 1_2, \mathrm{hd} \circ 2_2, \mathrm{conc} \circ \mathrm{tl} \circ 2_2] \circ [\mathrm{rev} \circ \mathrm{abs} \circ 3, \mathrm{cons} \circ [2, \mathrm{rev} \circ \mathrm{abs} \circ 1]]$$

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; [\mathrm{conc} \circ \mathrm{rev} \circ \mathrm{abs} \circ 3, 2, \mathrm{conc} \circ \mathrm{rev} \circ \mathrm{abs} \circ 1]$$

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; [\mathrm{conc} \circ \mathrm{abs} \circ \mathrm{revtree} \circ \mathrm{conc} \circ \mathrm{abs} \circ 3, 2, \mathrm{conc} \circ \mathrm{abs} \circ \mathrm{revtree} \circ \mathrm{conc} \circ \mathrm{abs} \circ 1]$$

by substituting for rev

$$= \mathrm{abs} \circ \mathrm{revtree} \circ \mathrm{conc}$$

$$= \mathrm{isempty} \rightarrow \mathrm{empty}; [\mathrm{revtree} \circ 3, 2, \mathrm{revtree} \circ 1]$$

(Recalling the comment above, more generally this equation would be an approximation, the fixed point we define here being one solution.)

In other words we have synthesised the following recursion equations for revtree

revtree (empty) = empty

revtree (node (t1, n, t2))

$$= \mathrm{node}(\mathrm{revtree}(\mathrm{t2}), n, \mathrm{revtree}(\mathrm{t1}))$$

Now the execution time of reverse(x) is of order the length of x, whereas revtree(conc(x)) can theoretically execute in time of order $\log_2$ of the length of x on parallel processors.

4.2 Representing lists as vectors

We consider a problem in which a list is represented sequentially as a vector with an offset to indicate the current position of the head, i.e. with type v-list = (vector × integer), the types of item in the list and vector being the same. It is required to determine the concrete version of the function which appends two abstract lists. The problem is shown in the following diagram.

$$\begin{tikzcd}
\text{list} \arrow[r, "\text{append}"] & \text{list} \arrow[d, "\text{abs}^{-1}"] \\
v\text{-list} \arrow[u, "\text{abs}"] \arrow[r, "\text{append}'"'] & v\text{-list}
\end{tikzcd}$$

Given abs and append, we need to find first conc = abs^{-1} and then append′, which will have the same recursive structure as append, as observed in Section 2.2. We will use a function-level version of the notation given by Darlington.[6] We already have the axioms for lists in the form of the FP sequence-axioms and just define

nil′ : → vector (the empty vector constructor function)

assign : vector × value × integer → vector

read : vector × integer → value

The type value is an arbitrary base type for the elements of the vector and of the list, for example integer. The axiom given for the vector type is

$$\mathrm{read}(\mathrm{assign}(v, i, j), k) = \begin{cases} i & \textbf{if } j = k \\ \mathrm{read}(v, k) & \textbf{otherwise} \end{cases}$$

This immediately implies that the inverse function of read is read^{-1} given by

$$\mathrm{read}^{-1} = [\mathrm{assign} \circ [?1, \mathrm{id}, ?2], ?2]$$

for logical function variables ?1, ?2.

We will also use another axiom for the vector data type, concerning the function assign. For integer $n > 0$, and functions x, v,

$$\ll \mathrm{assign} \circ [?1, x, \mathrm{add}(n) \circ 2 \circ v], 1 \circ v \gg$$
$$= \mathrm{assign} \circ [1 \circ v, x, \mathrm{add}(n) \circ 2 \circ v]$$

where x returns a 'value' and v returns an object of type v-list. This is consistent with the axioms given for vectors and intuitively correct because

$$\mathrm{read} \circ v = \mathrm{read} \circ \mathrm{assign} \circ [1 \circ v, x, \mathrm{add}(n) \circ 2 \circ v] \; \forall n \neq 0.$$

In other words, if an assignment is made in a position in the vector component of an object of type v-list which is not the head position (its second component), the result of reading the head item is unchanged.

It is also given that abs : vector × integer → list, is defined by

$$\mathrm{abs} = \mathrm{eq0} \circ 2 \rightarrow \mathbf{nil}; \mathrm{cons} \circ [\mathrm{read}, \mathrm{abs} \circ [1, \mathrm{sub1} \circ 2]]$$

or $\mathrm{abs} = \mathrm{eq0} \circ 2 \rightarrow \mathbf{nil}; H\mathrm{abs}$ for appropriately defined functional H.

We must now find abs^{-1} (i.e. conc) from abs, and hence cons′, hd′, tl′, the versions of the primitives cons, hd and tl defined on the vector representation of lists. Application of the standard FP laws will then yield the required defining expression for append′. Indeed, in this way the answer obtained by Darlington follows by purely mechanistic (and mechanisable) means. In particular, the where-abstractions are generated quite naturally using constructions (created during the formation of inverse functions), which are accessed by (compositions of) selector functions. Of course, the choices of the new axiom and of the definition of read^{-1} are not fully automatic, but these are the only 'Eureka-steps', and the properties of the target type must in any case be specified in some way by the programmer.

For the primitive function hd, since list items in each representation are identical, we have a 'commutative triangle' and

$$\mathrm{hd}' = \mathrm{hd} \circ \mathrm{abs} = \mathrm{read}$$

(strictly, $\mathrm{hd}' = \mathrm{eq0} \circ 2 \rightarrow \bot; \mathrm{read}$, but we assume read returns ⊥ when applied to an empty vector). The selector function tl has type list $\to$ list, so that $\mathrm{tl}' = \mathrm{conc} \circ \mathrm{tl} \circ \mathrm{abs} = \mathrm{conc} \circ \mathrm{abs} \circ [1, \mathrm{sub1} \circ 2] = [1, \mathrm{sub1} \circ 2]$.

To synthesise cons′, we will need the expression for conc (which has serendipitously cancelled so far). By direct application of the inversion method, we obtain

$$
\begin{aligned}
(H\,\mathrm{abs})^{-1} &= [\mathrm{read}, \mathrm{abs} \circ [1, \mathrm{sub1} \circ 2]]^{-1} \circ \mathrm{cons}^{-1} \quad \text{by (I 1)}\\
&= \ll \mathrm{read}^{-1} \circ 1, [1, \mathrm{sub1} \circ 2]^{-1} \circ \mathrm{abs}^{-1} \circ 2 \gg \\
&\quad \circ [\mathrm{hd}, \mathrm{tl}] \quad \text{by (A 6) and (I 2)}\\
&= \ll \mathrm{read}^{-1} \circ 1, \ll 1^{-1} \circ 1, 2^{-1} \circ \mathrm{add1} \circ 2 \gg \\
&\quad \circ \mathrm{conc} \circ 2 \gg \circ [\mathrm{hd}, \mathrm{tl}] \quad \text{by (I 2)}\\
&= \ll \mathrm{read}^{-1} \circ 1, [1, \mathrm{add1} \circ 2] \circ \mathrm{conc} \circ 2 \gg \\
&\quad \circ [\mathrm{hd}, \mathrm{tl}] \quad \text{by (T 1)}\\
&= \ll [\mathrm{assign} \circ [?1, \mathrm{id}, ?2] \circ 1, [1,\\
&\quad \mathrm{add1} \circ 2] \circ \mathrm{conc} \circ 2 \gg \circ [\mathrm{hd}, \mathrm{tl}]\\
&\quad \text{substituting for read}^{-1}\\
&= \ll [\mathrm{assign} \circ [?1, 1, ?2], ?2], [1 \circ \mathrm{conc} \circ 2,\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2] \gg \circ [\mathrm{hd}, \mathrm{tl}]\\
&= \ll [\mathrm{assign} \circ [?1, 1, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2],\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2], [1 \circ \mathrm{conc} \circ 2,\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2] \gg \circ [\mathrm{hd}, \mathrm{tl}]\\
&\quad \text{by axioms (A 3) and (A 1)}\\
&= [\ll \mathrm{assign} \circ [?1, 1, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2].\\
&\quad 1 \circ \mathrm{conc} \circ 2 \gg, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2] \circ [\mathrm{hd}, \mathrm{tl}]\\
&\quad \text{by (A 3)}\\
&= [\mathrm{assign} \circ [1 \circ \mathrm{conc} \circ 2, 1, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2],\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ 2] \circ [\mathrm{hd}, \mathrm{tl}]\\
&\quad \text{substituting for } v = \mathrm{conc} \circ 2,\\
&\quad x = 1 \circ 1, n = 1 \text{ in the axiom}\\
&= [\mathrm{assign} \circ [1 \circ \mathrm{conc} \circ \mathrm{tl}, \mathrm{hd},\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}], \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}]
\end{aligned}
$$

Hence by the main inversion result, (I 3), we obtain the following definition for conc:

$$
\begin{aligned}
\mathrm{conc} = \mathrm{null} \to [\mathbf{nil}', \mathbf{0}]; [&\mathrm{assign} \circ [1 \circ \mathrm{conc} \circ \mathrm{tl},\\
&\mathrm{hd}, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}], \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}]
\end{aligned}
$$

Similarly, recalling the commutative square for cons above,

$$
\begin{aligned}
\mathrm{cons}' = [&\mathrm{assign} \circ [1 \circ \mathrm{conc} \circ \mathrm{tl}, \mathrm{hd}, \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}],\\
&\mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{tl}] \circ \mathrm{cons} \circ [1, \mathrm{abs} \circ 2]
\end{aligned}
$$

(using the facts that $\mathrm{null} \circ \mathrm{cons} = \mathbf{F}$ and that id maps (list) values to (vector) values, giving a mapping $[1, \mathrm{abs} \circ 2]$ for a value-vector pair)

$$
\begin{aligned}
&= [\mathrm{assign} \circ [1 \circ \mathrm{conc} \circ \mathrm{abs} \circ 2, 1,\\
&\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{abs} \circ 2],\\
&\quad\quad \mathrm{add1} \circ 2 \circ \mathrm{conc} \circ \mathrm{abs} \circ 2]\\
&= [\mathrm{assign} \circ [1 \circ 2, 1, \mathrm{add1} \circ 2 \circ 2], \mathrm{add1} \circ 2 \circ 2]
\end{aligned}
$$

Finally, since the range of null and null′ are the same, i.e. the boolean type,

$$
\begin{aligned}
\mathrm{null}' = \mathrm{null} \circ \mathrm{abs} &= \mathrm{eq0} \circ 2 \to \mathrm{null} \circ \mathbf{nil};\\
&\quad \mathrm{null} \circ \mathrm{cons} \circ [\mathrm{read}, \mathrm{abs} \circ [1, \mathrm{sub1} \circ 2]]\\
&= \mathrm{eq0} \circ 2 \to \mathbf{T}; \mathbf{F} = \mathrm{eq0} \circ 2
\end{aligned}
$$

Thus we obtain by the result in Section 2.2

$$
\begin{aligned}
\mathrm{append}' &= \mathrm{eq0} \circ 2 \circ 1 \to 2; \mathrm{cons}' \circ [\mathrm{hd}' \circ 1,\\
&\qquad \mathrm{append}' \circ [\mathrm{tl}' \circ 1, 2]]\\
&= \mathrm{eq0} \circ 2 \circ 1 \to 2; [\mathrm{assign} \circ [1 \circ 2, 1,\\
&\qquad \mathrm{add1} \circ 2 \circ 2], \mathrm{add1} \circ 2 \circ 2] \circ [\mathrm{read} \circ 1,\\
&\qquad \mathrm{append}' \circ [[1, \mathrm{sub1} \circ 2] \circ 1, 2]]
\end{aligned}
$$

This is the equation required, which perhaps becomes clearer if we write $[u,j]$ to denote $\mathrm{append}' \circ [[1, \mathrm{sub1} \circ 2] \circ 1, 2]$, and simplify the else-part expression using FP axioms to get

$$
\begin{aligned}
\mathrm{append}' = \mathrm{eq0} \circ 2 \circ 1 \to 2;&\\
&[\mathrm{assign} \circ [u, \mathrm{read} \circ 1, \mathrm{add1} \circ j], \mathrm{add1} \circ j]
\end{aligned}
$$

where $[u,j] = \mathrm{append}' \circ [[1, \mathrm{sub1} \circ 2] \circ 1, 2]$.

This is now in precisely the form that appears in Ref. 6, i.e.

$$
\begin{aligned}
&\mathrm{append}'(<v_1, 0>, <v_2, i_2>) = <v_2, i_2>\\
&\mathrm{append}'(<v_1, i_1 + 1>, <v_2, i_2>)\\
&\quad = \mathrm{assign}(u, \mathrm{read}(v_1, i_1 + 1), j + 1)\\
&\quad\quad \textbf{where } <u, j> == \mathit{append}'(v_1, i_1>, <v_2, i_2>).
\end{aligned}
$$

5. TRANSFORMATION OF DATA TYPES FOR OPTIMISATION OF GIVEN OPERATIONS

Given a data type, we often require to optimise a particular set of its operations. After all, the purpose of any representation is to facilitate some computation. For example, having used a high-level representation of a data type such as 'list of numbers' we may wish to find an implementation that optimises the operation *minimum* on it. One possible solution is a list of numbers in ascending order. In this way the *minimum* operation would simply be implemented as *head*. This type of transformation requires greater deductive capabilities and in general (assuming such an optimisation exists) appears elusive. However, given the capacity to recognise the applicability of axioms such as (A 9*b*), i.e. the ability to detect statically that one predicate implies another, the function-level approach outlined below provides a solution that is mechanisable.

In general we would like to optimise $n \geqslant 1$ operations, $f_1, \ldots, f_n$ on a data type α, and to synthesise the function, conc_α, that maps the abstract type α to its concrete implementation type on which the optimised versions, f_i', of f_i will operate. The idea is to try to generate the conc function automatically so that most of the 'work' performed by the functions f_i' is included in it, so that we can then give new definitions for the f_i' that are inexpensive, representing the remainder of that 'work'.

Now, given f_i of type $\alpha \to \beta$ we can again make use of the rule $f_i' \circ \mathrm{conc}_\alpha = \mathrm{conc}_\beta \circ f_i$. We then have in general that $\mathrm{conc}_\alpha = f_i'^{-1} \circ \mathrm{conc}_{\beta i} \circ f_i$ for all i such that $f_i : \alpha \to \beta_i$, where β_i may be any type. Since all such equations must be satisfied simultaneously, there must be a substitution for the logical function variables in each right-hand side which renders them all equal. Thus we define

$$\mathrm{conc}_\alpha = \ll f_i'^{-1} \circ \mathrm{conc}_{\beta i} \circ f_i \,|\, f_i : \alpha \to \beta_i \gg$$

with obvious notation. In this way, given expressions for the concrete functions f_i' ($1 \leqslant i \leqslant n$) it may be possible to use the n-fold function-level unification as a *definition* definition for conc_α. However, note that although conc_α must satisfy this equation, it is not necessarily equal to its solution (i.e. its least fixed point).

The possible results of the unification now fall into three classes. (*a*) Failed, i.e. there is no substitution and so no optimised type can be found. (*b*) A result which contains logical function variables, in which case an optimised type exists but is not unique. (*c*) A result which contains no logical function variable, in which case there exists a unique optimised type, the result being the associated concretisation function.

In fact here we have introduced function inverses which may not be single-valued, and so in general we require the more complex apparatus of a general analysis of inverse functions. However, our present capabilities are sufficient for the following examples.

Example 1

Suppose we wish to optimise two operations on lists: 'last' which finds the last item and 'Rtail' which returns the list excluding its last element. These functions are defined on the abstract type list(α) by

$$\begin{aligned}&\text{last}(x::\text{nil}) = x\\ &\text{last}(x::y::1) = \text{last}(y::1)\\ &\text{Rtail}(x::\text{nil}) = \text{nil}\\ &\text{Rtail}(x::y::1) = x::\text{Rtail}(y::1)\end{aligned}$$

The definition of last and Rtail in FP is as follows:

$f_1 = \text{last} = \text{null}\circ\text{tl} \rightarrow \text{hd}; \text{last}\circ\text{tl}$

and

$$f_2 = \text{Rtail} = \text{null}\circ\text{tl} \rightarrow \textbf{nil}; \text{cons}\circ[\text{hd}, \text{Rtail}\circ\text{tl}]$$

Now suppose the concrete functions proposed for last and Rtail are specified as

$$f_1' = \text{last}' = \text{hd} \quad\text{and}\quad f_2' = \text{Rtail}' = \text{tl}$$

The synthesised representation function $\text{conc}_{\text{list}(\alpha)}$, abbreviated to conc, is then the result of the function-level unification of the right-hand sides of the equations derived for it from each of the function-pairs (f_i, f_i'), $i = 1, 2$.

The base case of the representation function is also specified, so as to preserve the null list, nil. The synthesised representation function, conc, is then defined by

$\text{conc}\circ\textbf{nil} = \textbf{nil}$ and

$$\begin{aligned}\text{conc} &= \lll f_1'^{-1}\circ f_1, f_2'^{-1}\circ\text{conc}\circ f_2 \ggg\\
&= \lll \text{null}\circ\text{tl}\rightarrow\text{hd}^{-1}\circ\text{hd}; \text{hd}^{-1}\circ\text{last}\circ\text{tl},\\ &\qquad \text{null}\circ\text{tl}\rightarrow\text{tl}^{-1}\circ\text{conc}\circ\textbf{nil};\\ &\qquad \text{tl}^{-1}\circ\text{conc}\circ\text{cons}\circ[\text{hd}, \text{Rtail}\circ\text{tl}]\ggg\\
&= \text{null}\circ\text{tl}\rightarrow \lll\text{hd}^{-1}\circ\text{hd}, \text{tl}^{-1}\circ\text{conc}\circ\textbf{nil}\ggg;\\ &\qquad \lll\text{hd}^{-1}\circ\text{last}\circ\text{tl}, \text{tl}^{-1}\circ\text{conc}\circ\text{cons}\circ[\text{hd},\\ &\qquad \text{Rtail}\circ\text{tl}]\ggg\\ &\qquad \text{by (A 9), since equal predicates}\\
&= \text{null}\circ\text{tl}\rightarrow\\ &\qquad \text{cons}\circ[\text{hd}, \textbf{nil}]; \text{cons}\circ[\text{last}\circ\text{tl}, \text{conc}\circ\text{cons}\circ[\text{hd},\\ &\qquad \text{Rtail}\circ\text{tl}]]\\ &\qquad \text{by (T 2) and since conc}\circ\textbf{nil} = \textbf{nil}\\
&= \text{null}\circ\text{tl}\rightarrow[\text{hd}]; \text{cons}\circ[\text{last}\circ\text{tl},\\ &\qquad \text{conc}\circ\text{cons}\circ[\text{hd}, \text{Rtail}\circ\text{tl}]]\\ &\qquad \text{by FP axiom}\end{aligned}$$

This synthesised function therefore forms the concrete data type by simply reversing the list.

Example 2

Suppose we wish to optimise two operations on lists of numbers: 'min', which finds the minimum item and 'deletemin', which deletes it. These functions are defined on the abstract type $\alpha = \text{list(num)}$ by

$$\begin{aligned}\min(x::\text{nil}) &= x\\ \min(x::y::1) &= \textbf{if } \min(y::1) \geqslant x \textbf{ then } x \textbf{ else } \min(y::1)\\ \text{delete}(x, y::1) &= \textbf{if } x = y \textbf{ then } 1 \textbf{ else } y::\text{delete}(x, 1)\\ \text{deletemin}(1) &= \text{delete}(\min(1), 1)\end{aligned}$$

It is intended to represent min by hd and deletemin by tl on the concrete type, i.e. $f_1' = \text{hd}, f_2' = \text{tl}$ as in Example 1. The definition of min and deletemin in FP is as follows:

$f_1 = \min = \text{null}\circ\text{tl}\rightarrow\text{hd}; \text{le}\circ[\text{hd}, \min\circ\text{tl}]\rightarrow\text{hd}; \min\circ\text{tl}$

$f_2 = \text{deletemin} = \text{delete}\circ[\min, \text{id}]$,

$$\begin{aligned}\text{where delete} &= \text{eq}\circ[1, \text{hd}\circ 2]\rightarrow\text{tl}\circ 2;\\ &\qquad \text{cons}\circ[\text{hd}\circ 2, \text{delete}\circ[1, \text{tl}\circ 2]], \text{ i.e.}\\
\text{deletemin} &= \text{eq}\circ[1, \text{hd}\circ 2]\circ[\min, \text{id}]\rightarrow\text{tl}\circ 2\circ[\min, \text{id}];\\ &\qquad \text{cons}\circ[\text{hd}\circ 2, \text{delete}\circ[1, \text{tl}\circ 2]]\circ[\min, \text{id}]\\
&= \text{eq}\circ[\min, \text{hd}]\rightarrow\text{tl};\\ &\qquad \text{cons}\circ[\text{hd}, \text{delete}\circ[\min, \text{tl}]]\end{aligned}$$

For convenience of notation, we write these as

$$\min = \text{null}\circ\text{tl}\rightarrow\text{hd}; H$$

and

$\text{deletemin} = \text{eq}\circ[\min, \text{hd}]\rightarrow\text{tl}; G$

for appropriately defined expressions H and G.

The base case of the representation function is also specified, so as to preserve the null list, nil. As in Example 1, the synthesised representation function, conc, is then defined by

$\text{conc}\circ\textbf{nil} = \textbf{nil}$ and

$$\begin{aligned}&\text{conc} \lll f_1'^{-1}\circ f_1, f_2'^{-1}\circ\text{conc}\circ f_2\ggg\\
&\quad= \lll\text{null}\circ\text{tl}\rightarrow\text{hd}^{-1}\circ\text{hd}; \text{hd}^{-1}\circ H, \text{eq}\circ\\ &\qquad [\min, \text{hd}]\rightarrow\text{tl}^{-1}\circ\text{conc}\circ\text{tl}; \text{tl}^{-1}\circ\text{conc}\circ G\ggg\\
&\quad= \text{null}\circ\text{tl}\rightarrow\lll\text{hd}^{-1}\circ\text{hd}, \text{tl}^{-1}\circ\text{conc}\circ\text{tl}\ggg;\\ &\qquad (\text{eq}\circ[\min, \text{hd}]\rightarrow\lll\text{hd}^{-1}\circ H,\\ &\qquad \text{tl}^{-1}\circ\text{conc}\circ\text{tl}\ggg; \lll\text{hd}^{-1}\circ H,\\ &\qquad \text{tl}^{-1}\circ\text{conc}\circ G\ggg) \quad \text{by (A 9}b\text{)},\\ &\qquad \text{since null}\circ\text{tl implies eq}\circ[\min, \text{hd}]\\
&\quad= \text{null}\circ\text{tl}\rightarrow\text{cons}\circ[\text{hd}, \text{conc}\circ\textbf{nil}]; (\text{eq}\circ[\min,\\ &\qquad \text{hd}]\rightarrow\lll\text{hd}^{-1}\circ H, \text{tl}^{-1}\circ\text{conc}\circ\text{tl}\ggg;\\ &\qquad \lll\text{hd}^{-1}\circ H, \text{tl}^{-1}\circ\text{conc}\circ\\ &\qquad G\ggg) \quad \text{by (T 2) and since null}\circ\text{tl}\\ &\qquad \text{implies tl} = \textbf{nil}\\
&\quad= \text{null}\circ\text{tl}\rightarrow[\text{hd}]; (\text{eq}\circ[\min, \text{hd}]\rightarrow\lll\text{hd}^{-1}\circ H,\\ &\qquad \text{tl}^{-1}\circ\text{conc}\circ\text{tl}\ggg; \lll\text{hd}^{-1}\circ H,\\ &\qquad \text{tl}^{-1}\circ\text{conc}\circ G\ggg) \quad \text{by FP}\\ &\qquad \text{axiom and since conc}\circ\textbf{nil} = \textbf{nil}\\
&\quad= \text{null}\circ\text{tl}\rightarrow[\text{hd}]; (\text{eq}\circ[\min, \text{hd}]\rightarrow\text{cons}\circ\\ &\qquad [H, \text{conc}\circ\text{tl}]; \text{cons}\circ[H, \text{conc}\circ G])\\ &\qquad \text{by (T 2) for both parts}\\ &\qquad \text{involving function-level unification}\end{aligned}$$

where H and G are the else parts of the definitions of min and deletemin respectively.

Our result is very similar to what an intelligent programmer might have designed for this purpose. Such a function might be:

$\text{conc}(\text{nil}) = \text{nil}$

$\text{conc}(x::\text{nil}) = x::\text{nil}$

$\text{conc}(x::y::z) = \min(x::y::z)::$

$\quad \text{conc}(\text{deletemin}(x::y::z))$

Here conc is single-valued, but suppose instead we only had the operation deletemin which we wished to optimise by tl. Then the concretisation function would be conc′ defined by

$$\text{conc}' = \text{tl}^{-1} \circ \text{conc}' \circ \text{deletemin}$$

where $\text{conc}' \circ \textbf{nil} = \textbf{nil}$

Thus

$$\text{conc}' = \text{cons} \circ [?1, \text{conc}' \circ \text{deletemin}]$$
$$= \text{cons} \circ [?1, \text{cons} \circ [?2, \ldots \text{cons} \circ [?n, \textbf{nil}] \ldots]$$

when applied to a list of length n.

In fact the concretisation function, conc′, is the most general, and here it loses all information as to the values of an abstract list's elements. This is actually quite acceptable, since we have not represented any selector function on our concrete type. If, for example, we had also wished to represent the function 'max' (giving the maximum element) by hd, we would have obtained $\text{conc}' = \text{cons} \circ [\max, \text{cons} \circ [\max, \ldots \text{cons} \circ [\max, \textbf{nil}] \ldots]$, or if we had required max to be represented by last we would have obtained $\text{conc}' = \text{cons} \circ [?1, \ldots \text{cons} \circ [?(n-1), \text{cons} \circ [\max, \textbf{nil}] \ldots]$, for lists of length n. In every example, the concrete version of a selector function returns the correct element from the concrete list, and also from the concrete versions of the lists obtained by successively deleting the minimum element. Note that although we can always choose some unique definition of conc, by appropriate assignments to the logical variables, the corresponding abstraction function, conc'^{-1}, is many-valued because of the destructive nature of the function deletemin. Our assumption of single-valuedness generalises satisfactorily here, but we are approaching the same problem as we noted in Section 2, concerning the representation of sets by lists.

6. CONCLUSION

The mechanical synthesis of efficient implementations of abstract data types as concrete types benefits greatly from an algebraic approach using a combinator-based analysis. Our results are based on the simple observation that a square of functions defined on the abstract and concrete domain types and range types must commute. Assuming the availability of the composition operation and of inverses for the given abstraction functions (mapping the concrete type to the abstract type), the required concrete version of a function defined on an abstract type can then be synthesised using the axioms of a functional algebra augmented with those relating to the theory of inverse functions and function-level unification. In this way we were able to construct, mechanically, concrete versions of functions defined on a programmer's abstract lists, which were represented efficiently by trees or by vectors. We also showed how particular concrete types could be synthesised so as to optimise certain given operations on the abstract types, for example 'minimum' and 'delete-minimum' defined on a list of numbers.

We observed, however, that our axiom set required extension by further axioms representing operations on the particular data types in question, and in the examples these were not necessarily expressed in a form that would normally be given by the programmer. This is certainly the area of the work which is the least mechanised, and some interface between the programming of data types and this type of transformation system should be investigated. One well-known way of representing a data type is a *formal theory*. The syntax of terms is determined by some algebra of operators over some carrier set of objects. Then the type-constructors (amongst other functions) are represented by operators and the carrier set represents the elements in a compound object of the type. A *theory* consists of an algebra together with a set of equations identifying terms, universally quantified over their variable. Using such a representation, we can choose equations appropriate to a particular kind of implementation. For example, if we have a parallel implementation in mind, we might choose to consider lists as composed of an element 'somewhere in the middle' with a sub-list appended on either side, giving a *symmetrical* view. For the reverse function on lists, we would then naturally include in our theory the equation

$$\text{reverse}\,(A <> <x> <> B) = \text{reverse}\,(B) <> <x> <> \text{reverse}\,(A)$$

(compare Section 4.1). This use of a theory brings the necessary axioms 'up front', although it is clear that much more research is necessary to develop a rigorous and practical scheme.

A much more substantial generalisation of this work would permit the abstraction and concretisation functions to be many-valued. For example, using such functions it may become possible to represent sets by lists in which elements may be duplicated, and to synthesise given set operations as corresponding (concrete) functions on lists – e.g. 'union' might be represented in terms of 'append'. This requires the full apparatus of the analysis of inverse functions where in general any function, single-valued or many-valued, is defined as a mapping between power domains.[9]

Few would deny the expressive power bestowed on any programming language by abstract data types, and efficient implementation is therefore crucial to permit exploitation of their full potential. The pilot transformation schemes introduced in this paper greatly support the enhancement of abstract data types as a software development tool.

REFERENCES

1. J. W. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM* **21** (8), 613–641 (1978).
2. J. W. Backus, The algebra of functional programs: function level reasoning, linear equations and extended definitions. In *Lecture Notes in Computer Science*, vol.

107, *Formalization of Programming Concepts*, pp. 1–43. Springer, New York (1981).
3. R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *JACM* **24** (1), 44–67 (1977).
4. R. M. Burstall and J. A. Goguen, Putting theories together to make specifications. *Proc. 5th IJCAI, Boston*, pp. 1045–1058 (1977).
5. J. Darlington, *The Synthesis of Implementations for Abstract Data Types*. Research Report, DOC 80/4, Department of Computing, Imperial College, London (1980).
6. J. Darlington, Program transformation. In *Functional Programming and its Applications*, edited J. Darlington, P. Henderson and D. A. Turner. Cambridge University Press (1982).
7. R. W. Floyd, Algorithm 245 treesort. *CACM* **7** (12) (1964).
8. J. V. Guttag, Abstract data types and the development of data structures. *CACM* **20**, 397–404 (1977).
9. P. G. Harrison and H. Khoshnevisan, *On the Synthesis of Inverse Functions*. Research Report, Department of Computing, Imperial College, London (1988).
10. C. A. R. Hoare, Proof of correctness of data representations. *Acta Informatica* **4**, 271–281 (1972).
11. R. B. Kieburtz and J. Shultis, Transformations of FP program schemes. In *Proceedings of ACM Conference on Functional Languages and Computer Architecture, Portsmouth, New Hampshire* (1981).
12. B. H. Liskov and S. N. Ziller, Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* **SE-1** (1), 7–19 (1975).
13. J. Vuillemin, *A Data Structure for Manipulating Priority Queues*. Internal Report, Département d'Informatique, Université de Paris Sud, France (1976).

Book Reviews

GERHARD BREWKA
Nonmonotonic Reasoning: Logical Foundation of Commonsense
Cambridge University Press. £19.50.
ISBN 0 521 38394 3.

This book discusses aspects of nonmonotonic reasoning at three levels: logic (proof- and/or model-theoretic accounts of nonmonotonic systems), implementation (automated theorem provers for nonmonotonic systems) and what the author terms pragmatic (rule-based nonmonotonic systems or nonmonotonic process systems).

It is at the first level where most of the influential approaches to nonmonotonic reasoning have been proposed. The book begins with a broad overview of many of these approaches. Some of the prominent features are (1) the clarity with which what many Artificial Intelligence (AI) workers consider to be difficult approaches are presented; and (2) the adequate accuracy of the presentation, which seems to have done justice to every approach cited in the book up to the point of controversy. Therefore, the book should prove extremely useful and valuable for the novice who needs an almost painless introduction to a very complicated area of research.

As for the researcher who is already in the field the book offers less, but it is still adequately valuable and interesting. Undoubtedly, there are some new ideas such as the author's proposal of preferred subtheories which are both stimulating and beneficial to a researcher.

The second level, to which only one chapter is dedicated, is as interesting as the first. The computational aspects of nonmonotonic logics are interestingly difficult to handle. More is expected at this level if one of the main objectives of the book is to show that theoretically sound nonmonotonic reasoning can be done efficiently and if the author believes that 'there will probably not be much progress in the development of formalisations, nor an increase in the trust in the existing ones, without programs, which can handle more realistic examples than those which have been studied so far'. It would have been extremely helpful if the book could have given or, at least, emphasised the need for an in-depth comparison of the different approaches cited at level one in terms of their computational properties.

For the pragmatic level, which is the most appealing to AI workers, the book presents an interesting attempt at formalising some type of 'truth maintenance systems' by providing model theoretical semantics.

The overall expository value of the different approaches and different levels of the book cannot be under-estimated. The book has been fairly successful in its main goal, which is to give a broad overview of the state of the art in different fields of research in the area of nonmonotonic reasoning. It does not give more than a vague orientation in the field. It regards the different research activities in the field as alternative ways of achieving sound theoretical foundations and efficient computation of nonmonotonic reasoning. This, in fact, is what one ideally wants to happen and believe, but the reality is something different. As earlier AI literature shows, workers at the pragmatic level had for a long period dismissed the idea of formalisation and sound theoretical foundation. What is surprising is that the book's message is that different activities, rather than different approaches to nonmonotonic reasoning, are ways of achieving a common goal.

N. OBEID
Colchester

DAVID LIGHTFOOT
Formal Specification using Z, Macmillan.
ISBN 0-333 544080. £13.99

The zealots of the 'formal methods' community will not like this book: it contains more natural language than dense mathematics, was not written using LaTeX, and is an attempt to teach a formal notation rather than show off the author's mathematical ability.

Lightfoot has concentrated on the system specification aspects of Z and has wisely ignored attempting to teach development, a topic which generates a large amount of concrete mathematics – even when the development is simple, such as the transformation from a sequence to a linked list.

The style of the book is pleasantly relaxed and the pace is not too fast. The author covers the important 80% of Z relevant to system specification in the traditional way: starting with logic and progressing, via sets, to functions and sequences. The only weakness of the book is that it eschews large examples. One of the major advantages of Z is its structuring facilities, particularly the way that the schema calculus enables system specifications to be presented in an incremental way. The author would have improved the book if he had included a final chapter describing a substantial example.

The potential audiences for this book are students on degree and HND courses, and staff in industry with a little mathematical knowledge who want a quick introduction to formal methods. Certainly, this is the first formal methods book that I have read which is suitable for HND students – albeit students in the final year of their course.

The formal methods community – and I count myself a member of it – seems to be in crisis. For the last ten years it has attracted quite a large amount of funding and formal methods are on the syllabus of every university computing department; yet progress has been pitifully slow, with the occasional embarrassing failure such as the Viper microprocessor acting as a punctuation mark. Certainly, the vibrations that I perceive from the Department of Trade and industry and the Science and Engineering Research Council are of increasing dissatisfaction with the subject over its lack of progress.

One of the reasons for this lack of progress is the small amount of good technology transfer materials that has been produced. The formal methods community has, over the last decade, contented itself with writing postgraduate-level textbooks and using them to teach undergraduates. Happily, this seems to be no longer true. The recent issue of *An Introduction to Formal Specification and Z* by Till, Potter and Sinclair, published by Prentice-Hall, has started to reverse the trend. This book continues it. The problem though, I suspect, is that it has been written five years too late.

D. INCE
Milton Keynes