Eureka Definitions for Free!

or

Disagreement Points for Fold/Unfold Transformations

Hanne Riis Nielson & Flemming Nielson
Computer Science Department
Aarhus University, Denmark

Abstract

The fold/unfold framework of Burstall and Darlington is a very powerful framework for transforming function definitions in the form of recursion equation schemes. This may be used to transform a function so as to improve the efficiency of its implementation. However, for this to work the user must supply so-called Eureka definitions and it may require some ingenuity to construct these. This paper shows that a class of these Eureka definitions can be derived in a rather systematic way.

1 Introduction

We shall begin by an example of the fold/unfold framework of Burstall and Darlington [1]. So consider the function

```
lookup (cons (pair x y) xys) z = IF eq x z THEN y ELSE lookup xys z
```

that takes a list of pairs as its first argument. It then takes a second argument and recursively looks for it as a first component in one of the pairs in the list and if successful returns the corresponding second component.

This function might be used in a compiler to lookup an identifier in the environment. Typically the identifiers will be known at compile-time whereas the actual values will not be known until run-time. This suggests that more efficient code could be generated if the lookup function was changed so that the compile-time part of the environment could be handled separately from the run-time part because then it would be a compile-time task to determine where some identifier occurs.

We therefore want to *transform* the program so as to *separate* the two components in the list of pairs. A curried version of the transformed lookup function may be written

```
lookup' (cons x xs) z (cons y ys) = IF eq x z THEN y ELSE lookup' xs z ys
```

As we shall now see this function can be derived in a systematic way using the *fold/unfold framework*.

The starting point will be a so-called *Eureka definition* and in general this is the step where human ingenuity seems most needed [1]:

```
lookup' xs z ys = lookup (zip (pair xs ys)) z
```

Here

```
zip (pair nil nil) = nil
```

```
zip (pair (cons x xs) (cons y ys)) = cons (pair x y) (zip (pair xs ys))
```

will construct a list of pairs from a pair of lists of equal length. We then get

```
lookup' (cons x xs) z (cons y ys)
  ≡ lookup (zip (pair (cons x xs) (cons y ys))) z      (Eureka definition)
  ≡ lookup (cons (pair x y) (zip (pair xs ys))) z      (unfold zip)
  ≡ IF eq x z THEN y ELSE lookup (zip (pair xs ys)) z  (unfold lookup)
  ≡ IF eq x z THEN y ELSE lookup' xs z ys              (fold lookup')
```

The purpose of this paper is to present a heuristics for how to construct (a class of) Eureka definitions. This centers around systematic type transformations and ways of identifying which type transformations may be useful.

So the starting point will be the type of the lookup function:

lookup: (LIST (PAIR α β)) $\rightarrow \alpha \rightarrow \beta$

Here α and β are type variables that can be instantiated with arbitrary types. As in SML [6], MIRANDA [13] and HASKELL [3] we have type constructors for user defined datatypes. The type constructors needed here may have been introduced by the following equations:

```
data LIST  $\alpha$  = nil + cons  $\alpha$  (LIST  $\alpha$ )
data PAIR  $\alpha$   $\beta$  = pair  $\alpha$   $\beta$ 
```

The type of the transformed function then is

lookup': (LIST α) $\rightarrow \alpha \rightarrow$ (LIST β) $\rightarrow \beta$

We shall see later that the corresponding Eureka definition can be produced systematically from the transformations on the types, i.e. from the transformation of the type of lookup into the type of lookup'.

2 Disagreement points

To prepare for this development we first need to identify *where* it might be profitable to apply a type transformation like the one that changed lookup into lookup'. This is facilitated by the tree representation of types. Here there will be three sorts of nodes in the trees that correspond to types:

- binary nodes labeled $\rightarrow$,
- k-ary nodes labeled by type constructors, and
- 0-ary nodes labeled by type variables.

Example: For the type LIST (PAIR STRING INT) $\rightarrow$ STRING $\rightarrow$ INT we get the tree:

■

Actually, we always consider a type in a certain context that contains the datatype definitions used in it. To model this we assume that there is a disagreement environment *daenv* that contains the necessary information (to be detailed below). Formally, this means that we always consider pairs consisting of a disagreement environment and a type (represented as a tree).

2.1 Consistently marked trees

To identify disagreement points we shall mark (some of) the nodes in type trees with information about which types should be *separated*. In this paper we shall use marks c and r for this purpose. In general one may have any number of marks but at least for the lookup function it seems sufficient with two marks as we only want to separate two types (using the intuition that one should live at compile-time and the other at run-time). The marks thus correspond to the different *stages* in [4].

We shall say that a tree is *consistently marked*, relative to a disagreement environment $daenv$, if

- all nodes are marked by either c or r ,
- if a node for the type constructor $tycon$ of arity k is marked d and its k sons are marked $d_1, \dots, d_k$ then $d_1 \dots d_k d \in daenv(tycon)$,
- if a node for the type variable $tyvar$ is marked d then $d \in daenv(tyvar)$,
- if a node for the function space constructor $\rightarrow$ is marked d and its two sons are marked d_1 and d_2 then $d_1 d_2 d \in daenv(\rightarrow)$.

Example: For the tree above we shall assume that $daenv(\text{STRING}) = \{c\}$ and $daenv(\text{INT}) = \{r\}$ in order to indicate that we want the **STRING** and **INT** components separated from one another. For the remaining type constructors we have $daenv(\text{PAIR}) = \{ccc, rrr\}$ and $daenv(\text{LIST}) = \{cc, rr\}$ to indicate that ‘the separations’ should be kept separate. We shall further assume that $daenv(\rightarrow) = \{ccc, rrr, rrc\}$ but no harm would result from using $daenv(\rightarrow) = \{ccc, rrr\}$. ■

The intention thus is that the mark of the root of a subtree indicates in which separation the corresponding subtype should be placed. The general use of $daenv(tycon) = \{c \dots c, r \dots r\}$ for a k -ary type constructor ($k \geq 1$) thus attempts to keep close subtypes together in the same separation if at all possible. Thus $daenv(\rightarrow)$ should contain at least ccc and rrr . By adding rrc one incorporates the intention that at compile-time one may manipulate code, i.e. functions that manipulate run-time entities, but it is beyond the scope of the present paper to go into the philosophical discussions of which choice is likely to be most useful in general.

However, given a type there is no guarantee that the corresponding tree can be made into a consistently marked tree. We shall therefore say that a node N in a partially marked tree is a *disagreement point* if

- each of the subtrees of N can be extended to consistently marked trees, but
- the subtree with root N cannot be extended to a consistently marked tree.

If the tree representation of an annotated type contains a disagreement point then that point indicates where a type transformation may be desired.

Example: For the example above we have two disagreement points. The node labeled **PAIR** is a disagreement point because its two subtrees can be marked consistently but the subtree itself cannot be. Also the lowest node in the tree labeled $\rightarrow$ is a disagreement point. The node labeled **LIST** is not a disagreement point since its subtree cannot be marked consistently. This demonstrates that disagreement points are situated as low in the tree as possible. ■

2.2 Disagreement point analysis

As observed in the above example it follows from the definition of disagreement points that they are as low in the tree as possible. This suggests that an algorithm for identifying disagreement points should perform a bottom-up traversal of the tree while checking the existence of consistently marked extensions of subtrees.

Consider the algorithm of Table 1. Here B is the set of disagreement markings that might apply for

```

INPUT:  a partially marked tree,  $T$ , and its disagreement environment,  $daenv$ 
OUTPUT: a disagreement point,  $N_0$ , if one exists
METHOD: paint all nodes of  $T$  white
        WHILE white nodes remain in  $T$  DO
            LET  $N_0$  be a (leftmost) leaf in the subtree of white nodes in  $T$ 
            LET  $N_1, \dots, N_k$  be the roots of the subtrees of  $N_0$ 
            CASE label of  $N_0$  OF  $tycon$ : LET  $B = daenv(tycon)$ 
                                    $tyvar$ : LET  $B = daenv(tyvar)$ 
                                    $\rightarrow$ :    LET  $B = daenv(\rightarrow)$ 
            LET  $C = B \cap \{d_1 \dots d_k d_0 \mid \forall i: \text{if } N_i \text{ is marked } d_i' \text{ then } d_i = d_i'\}$ 
            IF  $C = \emptyset$  THEN HALT (with  $N_0$  as disagreement point)
            FOR  $i = 0, 1, \dots, k$  DO
                IF  $\pi_i(C)$  is a singleton, e.g.  $\{d_i''\}$  THEN mark  $N_i$  in  $T$  with  $d_i''$ 
                paint  $N_0$  black

```

Table 1: Disagreement point analysis algorithm

as far as the disagreement environment, $daenv$, is concerned and C is the subset of B that agrees with the markings that have already been placed in the tree. The set $\pi_i(C)$ is obtained by selecting all i 'th components, i.e.

$$\pi_i(C) = \{d_i \mid d_1 \dots d_k d_0 \in C\}$$

and note that if N_i is marked with d_i' then $\pi_i(C)$ will be $\{d_i'\}$ so that the node N_i is simply remarked with $d_i'' = d_i'$.

The key insight needed to understand the algorithm is to observe that the following invariant holds at each entry to the WHILE-loop:

- all black subtrees may be extended to consistently marked trees, and
- if the root of a black subtree is presently unmarked then this can be done in at least two ways, in one of which the root is marked c and in the other the root is marked r .

Hence the node N_0 should be painted black and the subtrees rooted at each of N_i (for $i = 1, \dots, k$) remain black as no problems arise even if N_i was unmarked and is being marked (with the single element in $\pi_i(C)$).

The algorithm is inherently nondeterministic in the selection of candidates, N_0 , for disagreement points. As indicated in the algorithm this can be made deterministic by selecting the leftmost candidate. However, the algorithm only finds one disagreement point, e.g. the node labeled PAIR in the example of the previous subsection. To find all disagreement points the algorithm should do OUTPUT N_0 , rather than HALT, and the entire path from the root of T to N_0 must be painted black. The next candidate for N_0 now should be the (leftmost) leaf in the *forest* of white nodes in T .

3 Transformations on types

The presence of one or more disagreement points shows that the type is not fully separated. Furthermore, the disagreement points show where we may want to transform the type. After each transformation the intention is that the disagreement point has been removed, has been moved

closer to the root of the tree or has been moved from a left subtree to a right subtree. We will, however, not consider this formally in this paper.

3.1 Example: lookup

Based upon the type of the lookup function

$$\text{LIST (PAIR STRING INT)} \rightarrow \text{STRING} \rightarrow \text{INT}$$

we now show how the detection of disagreement points can be used to transform a type. As we saw above the type is represented as the tree

where we have indicated the markings on those subtrees that can be consistently marked. The disagreement point analysis identifies the roots of the subtypes PAIR STRING INT and STRING → INT as disagreement points. We shall proceed in three stages.

In the *first stage* we can do nothing to solve the problem indicated by the second disagreement point. However, matters can be improved slightly for the first disagreement point by replacing the subtype LIST (PAIR STRING INT) by the *larger*¹ type PAIR (LIST STRING) (LIST INT). The resulting type will be

$$\text{PAIR (LIST STRING) (LIST INT)} \rightarrow \text{STRING} \rightarrow \text{INT}$$

which has the tree representation

This tree has two disagreement points, the node labeled PAIR and the rightmost node labeled →. Thus the effect of the transformation on the type has been to move a disagreement point (the one that motivated the transformation) closer to the root of the tree while the other disagreement point(s) is unchanged.

In the *second stage* the overall function type is curried so it becomes

$$(\text{LIST STRING}) \rightarrow (\text{LIST INT}) \rightarrow \text{STRING} \rightarrow \text{INT}$$

with the tree representation

¹'Larger' in the sense of containing more elements.

Now only the rightmost node labeled $\rightarrow$ is a disagreement point and we see that a disagreement point (the one that motivated the transformation) has been removed.

In the *third stage* we will rearrange the order in which the functions take their arguments. So the type becomes

$$(\text{LIST STRING}) \rightarrow \text{STRING} \rightarrow (\text{LIST INT}) \rightarrow \text{INT}$$

The corresponding tree has no disagreement points and we are finished. (If we had used $daenv(\rightarrow) = \{ccc, rrr\}$ then the middle node labelled $\rightarrow$ would still be a disagreement point.)

3.2 A classification of transformations

The general pattern exhibited by the example above is that we transform one type, T , to another type, T' , and associated with this we have functions that map (a subset of the) values of type T into (a subset of the) values of type T' and vice versa. This transformation on values will be essential in the next section where we show how to construct the Eureka definitions corresponding to the type transformations.

A transformation is described by giving the two types T and T' and the two functions τ^+ and τ^- that transform the values:

$$T \begin{array}{c} \xrightarrow{\tau^+} \\ \xleftarrow{\tau^-} \end{array} T'$$

The transformations corresponding to the example of the previous subsection are given in Table 2.

Transformation 1:	$\text{LIST (PAIR } \alpha \beta) \begin{array}{c} \xrightarrow{\text{unzip}} \\ \xleftarrow{\text{zip}} \end{array} \text{PAIR (LIST } \alpha) (\text{LIST } \beta)$ <p> unzip nil = pair nil nil unzip (cons (pair x y) xys) = CASE unzip xys OF pair xs ys $\Rightarrow$ pair (cons x xs) (cons y ys) zip (pair nil nil) = nil zip (pair (cons x xs) (cons y ys)) = cons (pair x y) (zip (pair xs ys)) </p>
Transformation 2:	$(\text{PAIR } \alpha \beta) \rightarrow \gamma \begin{array}{c} \xrightarrow{\text{curry}} \\ \xleftarrow{\text{uncurry}} \end{array} \alpha \rightarrow \beta \rightarrow \gamma$ <p> curry f x y = f (pair x y) uncurry g (pair x y) = g x y </p>
Transformation 3:	$\alpha \rightarrow \beta \rightarrow \gamma \begin{array}{c} \xrightarrow{\text{reorder}} \\ \xleftarrow{\text{reorder}} \end{array} \beta \rightarrow \alpha \rightarrow \gamma$ <p> reorder f x y = f y x </p>

Table 2: Transformations: the lookup example

Intuitively we shall distinguish between four groups of transformations:

- the transformations that replace a type by an isomorphic type,
- the transformations that replace a type by a smaller type,
- the transformations that replace a type by a larger type, and

- the transformations that replace a subtype of one type by a subtype of another type.

The *first group* of transformations is fairly straightforward since it only allows isomorphisms. The two transformation functions on values are then the isomorphisms between the two types. As they are isomorphisms it will always be safe to use such a transformation and an example is Transformation 3 of Table 2.

The *second group* of transformations includes the *zip* direction of Transformation 1 of Table 2 whereas the *third group* includes the *unzip* direction of Transformation 1. However, by composing such transformations one easily ends up in the *fourth group*. As a more concrete example of a transformation in the fourth group consider matrix transpose where a matrix is represented as an element of LIST (LIST α): here matrices only correspond to lists of equal-length lists.

This demonstrates that in general we will have to accept that the transformation functions are partial functions unless we want to restrict ourselves to the rather limited class of transformations that fall within the first group.

3.3 Building transformation functions

So far we have considered transformations between entire trees:

$$T \begin{array}{c} \xrightarrow{\tau^+} \\ \xleftarrow{\tau^-} \end{array} T'$$

However, usually it will be the case that a subtree in T is replaced, leaving the remainder of the tree unchanged, i.e. $T = T_a[T_b]$ and $T' = T_a[T'_b]$ with

$$T_b \begin{array}{c} \xrightarrow{\tau_b^+} \\ \xleftarrow{\tau_b^-} \end{array} T'_b$$

As an example, the actual transformation in the first stage of Subsection 3.1 is on the subtype indicated by Transformation 1 of Table 2.

This then motivates defining τ^+ in terms of τ_b^+ (and τ_b^-). In the following we shall write

$$\tau: T \leftrightarrow T'$$

as an abbreviation for the pair of transformation functions

$$\tau = (\tau^+: T \rightarrow T', \tau^-: T' \rightarrow T)$$

The general idea will be to build τ by a bottom-up traversal over the tree using Reynolds' notion of a domain functor [12].

To do so it is helpful to borrow a few simple notions from category theory. In general, a functor consists of two mappings: one that works on 'types' (called objects) and one that works on 'functions' (called morphisms) and such that the functor laws hold (see later). We have seen that LIST, PAIR and $\rightarrow$ construct new types from given types. In a similar way they can be used to construct new functions from given functions. This is easiest for LIST and PAIR where a natural choice might be LIST $f = \text{map}(f)$ and PAIR $f g = f \times g$. Here $\text{map}(f)$ is the familiar construct for mapping the function f upon each component of a list and $f \times g = \lambda(x,y).(f(x),g(y))$ applies f and g componentwise. To allow for the functions f and g to be partial we shall say that $\text{map}(f)(z)$ is defined if and only if all invocations of f on the elements of z are defined and similarly $(f \times g)(x,y)$ is defined if and only if $f(x)$ and $g(y)$ are both defined.

As we consider transformation pairs it is actually more helpful to let LIST and PAIR operate on pairs of partial functions:

$$\text{LIST } \tau = (\text{map}(\tau^+), \text{map}(\tau^-))$$

$$\text{PAIR } \tau_1 \tau_2 = (\tau_1^+ \times \tau_2^+, \tau_1^- \times \tau_2^-)$$

This also allows to define the effect of function space:

$$\tau_1 \rightarrow \tau_2 = (\lambda f. \tau_2^+ \circ f \circ \tau_1^-, \lambda g. \tau_2^- \circ g \circ \tau_1^+)$$

Note that here the forward direction, i.e. the '+'-direction, of the transformation incorporates the '-'-direction of the τ_1 argument. (In categorical terms: function space is contravariant in its left argument.) This may be clarified by the following diagrams:

$$\begin{array}{ccc} T_1 & \xleftarrow{\tau_1^-} & T'_1 \\ \downarrow f & & \downarrow (\tau_1 \rightarrow \tau_2)^+(f) \\ T_2 & \xrightarrow{\tau_2^+} & T'_2 \end{array} \qquad \begin{array}{ccc} T_1 & \xrightarrow{\tau_1^+} & T'_1 \\ \downarrow (\tau_1 \rightarrow \tau_2)^-(g) & & \downarrow g \\ T_2 & \xleftarrow{\tau_2^-} & T'_2 \end{array}$$

We now have sufficient apparatus in order to define the transformation functions. So consider again the task of replacing a subtree T_b of a tree T by the subtree T'_b and let the transformation pair τ_b be supplied. To handle the remaining type variables and type constructors we shall assume that a *transformation environment*, $trenv$, gives information about how to transform. If a type constructor or type variable has arity k ($k \geq 0$) the transformation environment must indicate a function of functionality

$$(T_1 \leftrightarrow T'_1) \times \cdots \times (T_k \leftrightarrow T'_k) \rightarrow (T \leftrightarrow T')$$

where $T = tycon\ T_1 \cdots T_k$ and $T' = tycon\ T'_1 \cdots T'_k$ and similarly for the type variables. We then proceed by induction over the tree as follows:

- If the node is the root of the subtree (T_b) to be replaced (by the subtree T'_b) we set $\tau = \tau_b$.
- If the node is labeled by a type variable, *tyvar*, we set $\tau = trenv(tyvar)$.
- If the node is labeled by a type constructor, *tycon*, and $\tau_1, \dots, \tau_k$ are the transformation pairs constructed for its subtrees we put $\tau = trenv(tycon)(\tau_1, \dots, \tau_k)$.
- If the node is labeled by $\rightarrow$, and τ_1 and τ_2 are the transformation pairs constructed for the subtrees we put $\tau = \tau_1 \rightarrow \tau_2$.

Example: Consider the first transformation in Subsection 3.1. As indicated in Table 2, the subtree T_b corresponding to the subtype LIST (PAIR STRING INT) is replaced by the subtree T'_b corresponding to PAIR (LIST STRING) (LIST INT). The transformation pair $\tau_b : T_b \leftrightarrow T'_b$ accomplishing this is (unzip, zip). We next define $\tau : T \leftrightarrow T'$, where T is the tree corresponding to the original overall type LIST (PAIR STRING INT) $\rightarrow$ STRING $\rightarrow$ INT and T' is the tree corresponding to the new overall type PAIR (LIST STRING) (LIST INT) $\rightarrow$ STRING $\rightarrow$ INT. For this we will assume that the transformation environment, $trenv$, specifies $trenv(\text{STRING}) = \text{id}$ and $trenv(\text{INT}) = \text{id}$ where $\text{id}^+(f) = f$ and $\text{id}^-(g) = g$. We then obtain

$$\tau = (\text{unzip}, \text{zip}) \rightarrow (\text{id} \rightarrow \text{id})$$

and this amounts to

$$\tau^+(f) = (\text{id} \rightarrow \text{id})^+ \circ f \circ (\text{unzip}, \text{zip})^- = f \circ \text{zip}$$

$$(\text{and } \tau^-(g) = (\text{id} \rightarrow \text{id})^- \circ g \circ (\text{unzip}, \text{zip})^+ = g \circ \text{unzip}).$$

■

4 Construction of Eureka definitions

So far we have seen how the disagreement points can guide the transformation of the *type* of a function. But we have yet to see how the *function definition* itself can be changed. In fact each of the stages in the transformation on the type gives rise to parts of an Eureka definition that will be used to transform the function definition using the fold/unfold framework of Burstall and Darlington.

4.1 Example: lookup

For the example of Subsection 3.1 the *first stage* of the transformation on the type was to replace the list of pairs with a pair of lists. As a result the type was changed from $\text{LIST (PAIR STRING INT)} \rightarrow \text{STRING} \rightarrow \text{INT}$ to $\text{PAIR (LIST STRING) (LIST INT)} \rightarrow \text{STRING} \rightarrow \text{INT}$ and in the previous subsection we saw that the corresponding transformation on values is

$$\tau^+(f) = (\text{id} \rightarrow \text{id})^+ \circ f \circ (\text{unzip}, \text{zip})^- = f \circ \text{zip}$$

The corresponding Eureka definition therefore becomes

$$\text{lookup}_1 = \tau^+(\text{lookup})$$

which is equivalent to $\text{lookup}_1 (\text{pair } xs \ ys) \ z = \text{lookup} (\text{zip} (\text{pair } xs \ ys)) \ z$.

The *second stage* of the transformation on the type is currying. We shall replace the overall type $\text{PAIR (LIST STRING) (LIST INT)} \rightarrow \text{STRING} \rightarrow \text{INT}$ by $(\text{LIST STRING}) \rightarrow (\text{LIST INT}) \rightarrow \text{STRING} \rightarrow \text{INT}$. This is a direct application of one of the transformations of Table 2 and the transformation on values is therefore

$$\tau^+(f) = \text{curry} (f)$$

The corresponding Eureka definition becomes

$$\text{lookup}_2 = \text{curry} (\text{lookup}_1)$$

which is equivalent to $\text{lookup}_2 \ xs \ ys \ z = \text{lookup}_1 (\text{pair } xs \ ys) \ z$.

Finally, the *third stage* of the transformation on the type rearranges the order of the parameters. The type $(\text{LIST STRING}) \rightarrow (\text{LIST INT}) \rightarrow \text{STRING} \rightarrow \text{INT}$ is changed to $(\text{LIST STRING}) \rightarrow \text{STRING} \rightarrow (\text{LIST INT}) \rightarrow \text{INT}$. The construction of Subsection 3.3 then gives that the transformation on values is $\tau = \text{id} \rightarrow (\text{reorder}, \text{reorder})$ which means that

$$\tau^+(f)(x) = \text{reorder} (f \ x)$$

The corresponding Eureka definition becomes

$$\text{lookup}_3 \ xs = \text{reorder} (\text{lookup}_2 \ xs)$$

which is equivalent to $\text{lookup}_3 \ xs \ z \ ys = \text{lookup}_2 \ xs \ ys \ z$.

Putting these definitions together we get

$$\text{lookup}_3 \ xs \ z \ ys = \text{lookup} (\text{zip} (\text{pair } xs \ ys)) \ z$$

which is the Eureka definition used in the Introduction. Clearly this Eureka definition could also be obtained as $\text{lookup}_3 = \tau^+(\text{lookup})$ where

$$\tau^+(f)(x) = \text{reorder} (\text{curry} (f \circ \text{zip}) \ x)$$

is the composition of the three transformations functions from the transformation stages.

4.2 The general pattern

So far we have derived Eureka definitions without paying much attention to the nature of the transformation functions (τ^+): whether they are isomorphisms, whether they have left or right inverses or whether they are actually defined on their arguments.

To allow for transformation pairs that are pairs of partial functions we shall regard the simplification steps, e.g. $e_1 \equiv e_2$ of the fold/unfold framework as meaning that one side is defined if and only if the other is.

Consider now a transformation pair

$$T \begin{array}{c} \xrightarrow{\tau^+} \\ \xleftarrow{\tau^-} \end{array} T'$$

and an application

$$\tau^+(e)$$

As a *first attempt* at defining the existence of this application one might try to simplify $\tau^+(e)$ to an expression e' , i.e. $\tau^+(e) \equiv \dots \equiv e'$, such that e' only contains applications of transformation functions (τ_i^+ or τ_i^-) in contexts where it can be verified that they will be defined upon their particular arguments. This is too weak, however, as this would not be able to validate the first transformation on the lookup function of Subsection 4.1, because $\tau^+(\text{lookup}) = \text{lookup} \circ \text{zip}$ and when applied to a pair of lists of unequal length the transformations function $\tau_1^+ = \text{zip}$ will not be defined.

To *remedy* this we may note that the existence of $\text{lookup} \circ \text{zip}$ above is only really of interest when the argument is of a form that could have been produced by unzip . This observation then motivates the following attempt at defining when an application $\tau^+(e)$ is *deemed to exist*: Whenever $\tau^-(\tau^+(e))$ can be simplified to an expression e' , i.e. $\tau^-(\tau^+(e)) \equiv \dots \equiv e'$, such that e' only contains applications of transformation functions (τ_i^+ or τ_i^-) in contexts where it can be verified that they will be defined upon their particular arguments. Furthermore, the application $\tau^+(e)$ is *deemed to be safe* whenever e' above can be taken to be e .

Example: Consider the transformation upon the lookup function of the previous subsection. There

$$\tau = (\text{id} \rightarrow (\text{reorder}, \text{reorder})) \circ (\text{curry}, \text{uncurry}) \circ ((\text{unzip}, \text{zip}) \rightarrow (\text{id} \rightarrow \text{id}))$$

where composition of pairs of transformation pairs is given by $\tau_1 \circ \tau_2 = (\tau_1^+ \circ \tau_2^+, \tau_2^- \circ \tau_1^-)$. It follows that $\tau^+(f) = \text{reorder} \circ \text{curry} (f \circ \text{zip})$ and $\tau^-(g) = \text{uncurry} (\text{reorder} \circ g) \circ \text{unzip}$. To validate the application $\tau^+(\text{lookup})$ we calculate

$$\begin{aligned} \tau^-(\tau^+(\text{lookup})) &\equiv \text{uncurry} (\text{reorder} \circ \text{reorder} \circ \text{curry} (\text{lookup} \circ \text{zip})) \circ \text{unzip} \\ &\equiv \text{uncurry} (\text{curry} (\text{lookup} \circ \text{zip})) \circ \text{unzip} \\ &\equiv \text{lookup} \circ \text{zip} \circ \text{unzip} \\ &\equiv \text{lookup} \end{aligned}$$

so that the application is deemed to be safe (as well as to exist). ■

There would seem to be little interest in an application $\tau^+(e)$ that is deemed to exist but not deemed to be safe. This motivates defining a transformation pair $\tau = (\tau^+, \tau^-)$ to be a *partial isomorphism* if $\tau^-(\tau^+(x)) = x$ whenever the lefthand side is defined and, to allow for change of direction, $\tau^+(\tau^-(y)) = y$ whenever the lefthand side is defined. Thus τ will be a partial isomorphism if and only if (τ^-, τ^+) is.

All of the transformations in Table 2 are partial isomorphisms. Furthermore the property of being a partial isomorphism is being preserved when transformation functions are built in the way indicated in the previous section. For this result to hold we need that LIST, PAIR and $\rightarrow$ are so-called *definedness preserving functors*. We shall only explain the conditions for LIST as the other cases are similar. First we need the functor laws:

$\text{LIST id} = \text{id}$

$\text{LIST } (\tau_1 \circ \tau_2) = (\text{LIST } \tau_1) \circ (\text{LIST } \tau_2)$

Next we need the definedness preserving condition:

τ is a partial isomorphism $\Rightarrow \text{LIST } \tau$ is a partial isomorphism

By way of digression one may note the close relationship between these definedness preserving functors and what is often called locally monotonic functors: under suitable assumptions the condition that τ is a partial isomorphism may be reformulated as $\tau^- \circ \tau^+ \sqsubseteq \text{id}$ and $\tau^+ \circ \tau^- \sqsubseteq \text{id}$ and the local monotonicity condition

$\tau_1 \sqsubseteq \tau_2 \Rightarrow \text{LIST } \tau_1 \sqsubseteq \text{LIST } \tau_2$

(where $\tau_1 \sqsubseteq \tau_2$ means $\tau_1^+ \sqsubseteq \tau_2^+ \wedge \tau_1^- \sqsubseteq \tau_2^-$) and the functor laws then imply the definedness preserving condition.

Finally, we may observe that it has been the intention throughout that the transformation pairs only change the overall structure of the types but not the actual data buried there. As an example `zip` applied to a pair of lists of integers rearranges the order of the integers but does not actually change the integers, e.g. by adding 27 which is then subtracted by `unzip`. So far this intention has not been made explicit. To do so we may use a well-known remedy from category theory: that the transformations between the functors, e.g. from $\text{LIST PAIR } \alpha \beta$ to $\text{PAIR (LIST } \alpha) (\text{LIST } \beta)$, must be natural transformations. We shall therefore say that a transformation pair is a *natural partial isomorphism* if it is a partial isomorphism as detailed above and if furthermore the naturality squares commute. In case of the transformation from $\text{LIST PAIR } \alpha \beta$ to $\text{PAIR (LIST } \alpha) (\text{LIST } \beta)$ this demands that

$$\begin{array}{ccc}
 \text{LIST PAIR } \alpha \beta & \xleftarrow{\text{zip}} & \text{PAIR LIST } \alpha \text{ LIST } \beta \\
 \downarrow \text{LIST PAIR } f g & & \downarrow \text{PAIR LIST } f \text{ LIST } g \\
 \text{LIST PAIR } \alpha' \beta' & \xleftarrow{\text{zip}} & \text{PAIR LIST } \alpha' \text{ LIST } \beta'
 \end{array}$$

$$\begin{array}{ccc}
 \text{LIST PAIR } \alpha \beta & \xrightarrow{\text{unzip}} & \text{PAIR LIST } \alpha \text{ LIST } \beta \\
 \downarrow \text{LIST PAIR } f g & & \downarrow \text{PAIR LIST } f \text{ LIST } g \\
 \text{LIST PAIR } \alpha' \beta' & \xrightarrow{\text{unzip}} & \text{PAIR LIST } \alpha' \text{ LIST } \beta'
 \end{array}$$

both commute, i.e. we have the equalities $(\text{LIST } (\text{PAIR } f g)) \circ \text{zip} = \text{zip} \circ \text{PAIR } (\text{LIST } f) (\text{LIST } g)$ and $\text{unzip} \circ \text{LIST } (\text{PAIR } f g) = (\text{PAIR } (\text{LIST } f) (\text{LIST } g)) \circ \text{unzip}$.

4.3 Example: update

To illustrate the applicability of the technique for constructing Eureka definitions we shall consider yet another example:

```

update nil x y = cons (pair x y) nil
update (cons (pair x' y') xys) x y = IF eq x x' THEN cons (pair x y) xys
                                   ELSE cons (pair x' y') (update xys x y)

```

It has the type

$$\text{LIST}(\text{PAIR } \alpha\beta) \rightarrow \alpha \rightarrow \beta \rightarrow \text{LIST}(\text{PAIR } \alpha\beta)$$

We shall assume that the binding time environment $btenv$ has $btenv(\alpha) = \{c\}$ and $btenv(\beta) = \{r\}$; this means that (as in the lookup example) the first components of the pairs are known early and the second components are known late. We proceed in six stages.

In the *first stage*, disagreement point analysis identifies two disagreement points: the two nodes labeled PAIR. In both cases we shall apply Transformation 1 of Table 2 and the transformation pair becomes

$$\tau_1 = (\text{unzip}, \text{zip}) \rightarrow \text{id} \rightarrow \text{id} \rightarrow (\text{unzip}, \text{zip})$$

The resulting type will be

$$\text{PAIR}(\text{LIST } \alpha) (\text{LIST } \beta) \rightarrow \alpha \rightarrow \beta \rightarrow \text{PAIR}(\text{LIST } \alpha) (\text{LIST } \beta)$$

From τ_1 we get the transformation functions

$$\tau_1^+ f (\text{pair } xs \ ys) \ x \ y = \text{unzip} (f (\text{zip} (\text{pair } xs \ ys)) \ x \ y)$$

$$\tau_1^- g \ xys \ x \ y = \text{zip} (g (\text{unzip } xys) \ x \ y)$$

Using that $\text{zip} \circ \text{unzip} = \text{id}$ it is easy to verify that $\tau_1^-(\tau_1^+(f)) = f$ for all f and thereby that $\tau_1^+(f)$ is deemed to be safe. We now use the Eureka definition

$$\begin{aligned} \text{update}_1 (\text{pair } xs \ ys) \ x \ y &= \tau_1^+(\text{update}) (\text{pair } xs \ ys) \ x \ y \\ &= \text{unzip} (\text{update} (\text{zip} (\text{pair } xs \ ys)) \ x \ y) \end{aligned}$$

Using the fold/unfold framework it is straightforward to get

$$\begin{aligned} \text{update}_1 (\text{pair nil nil}) \ x \ y &\equiv \text{pair} (\text{cons } x \ \text{nil}) (\text{cons } y \ \text{nil}) \\ \text{update}_1 (\text{pair} (\text{cons } x' \ xs) (\text{cons } y' \ ys)) \ x \ y & \\ \equiv \text{IF } \text{eq } x \ x' \ \text{THEN CASE } \text{unzip} (\text{zip} (\text{pair } xs \ ys)) \ \text{OF} & \\ \quad \text{pair } xs' \ ys' \Rightarrow \text{pair} (\text{cons } x \ xs') (\text{cons } y \ ys') & \\ \text{ELSE CASE } \text{unzip} (\text{update} (\text{zip} (\text{pair } xs \ ys)) \ x \ y) \ \text{OF} & \\ \quad \text{pair } xs' \ ys' \Rightarrow \text{pair} (\text{cons } x' \ xs') (\text{cons } y' \ ys') & \end{aligned}$$

Now a fold step can be used to replace $\text{unzip} (\text{update} (\text{zip} (\text{pair } xs \ ys)) \ x \ y)$ by $\text{update}_1 (\text{pair } xs \ ys) \ x \ y$. Since $\tau_1^+(\text{update})$ is deemed to be safe we can also replace $\text{unzip} (\text{zip} (\text{pair } xs \ ys))$ by $\text{pair } xs \ ys$ although $\text{unzip} \circ \text{zip} = \text{id}$ does *not* hold in general. As expressed in the definition of 'deemed to be safe' the reason is that we shall only require update and update_1 to be equivalent when update_1 is supplied with a pair of lists of *equal length*. To be a bit more formal we should write $e_1 \equiv e_2 \ (\text{mod } \tau_1^-)$ to express that the equivalence is *modulo* τ_1^- . To define this notion we should prefer to write e.g.

$$\text{update}_1 = \lambda(\text{pair nil nil}).\lambda x.\lambda y.\text{pair} (\text{cons } x \ \text{nil}) (\text{cons } y \ \text{nil})$$

rather than

$$\text{update}_1 (\text{pair nil nil}) \ x \ y = \text{pair} (\text{cons } x \ \text{nil}) (\text{cons } y \ \text{nil})$$

because then $e_1 \equiv e_2 \ (\text{mod } \tau_1^-)$ would amount to $\tau_1^-(e_1) \equiv \tau_1^-(e_2)$. We arrive at the following definition

$$\begin{aligned} \text{update}_1 (\text{pair} (\text{cons } x' \ xs) (\text{cons } y' \ ys)) \ x \ y & \\ \equiv \text{IF } \text{eq } x \ x' \ \text{THEN } \text{pair} (\text{cons } x \ xs) (\text{cons } y \ ys) & \\ \text{ELSE CASE } \text{update}_1 (\text{pair } xs \ ys) \ x \ y \ \text{OF} & \\ \quad \text{pair } xs' \ ys' \Rightarrow \text{pair} (\text{cons } x' \ xs') (\text{cons } y' \ ys') & \end{aligned}$$

In the *second stage* we observe that the type

Transformation 4:	$\alpha \rightarrow (\text{PAIR } \beta \gamma) \xrightleftharpoons[\text{tofunc}]{\text{topair}} \text{PAIR } (\alpha \rightarrow \beta) (\alpha \rightarrow \gamma)$
	$\begin{aligned} \text{topair } f &= \text{pair } g \ h \text{ WHERE } g \ x = \text{CASE } f \ x \text{ OF pair } u \ v \Rightarrow u \\ &\quad \text{WHERE } h \ x = \text{CASE } f \ x \text{ OF pair } u \ v \Rightarrow v \\ \text{tofunc } (\text{pair } g \ h) &= f \text{ WHERE } f \ x = \text{pair } (g \ x) (h \ x) \end{aligned}$
Transformation 5:	$\alpha \rightarrow \beta \xrightleftharpoons[\text{recall}]{\text{forget}} \beta$
	$\begin{aligned} \text{forget } f &= f \ ? \\ &\quad \text{where } ? \text{ is some element of type } \alpha \\ \text{recall } x &= f \text{ WHERE } f \ y = x \end{aligned}$

Table 3: Transformations: the update example

$$\text{PAIR } (\text{LIST } \alpha) (\text{LIST } \beta) \rightarrow \alpha \rightarrow \beta \rightarrow \text{PAIR } (\text{LIST } \alpha) (\text{LIST } \beta)$$

still has two disagreement points: the two nodes labeled PAIR in the tree representation. For the leftmost disagreement point we shall use Transformation 2 of Table 2 and for the rightmost disagreement point we use that $\alpha \rightarrow \text{PAIR } \beta \gamma$ can be replaced by $\text{PAIR } (\alpha \rightarrow \beta) (\alpha \rightarrow \gamma)$. This transformation (see Table 3) belongs to the first group of transformations described in Subsection 4.2 provided that PAIR and $\rightarrow$ are interpreted either as cartesian product and continuous function space or as smash product and strict continuous function space. The transformation pair is

$$\tau_2 = (\text{id} \rightarrow \text{id} \rightarrow \text{id} \rightarrow (\text{topair}, \text{tofunc})) \circ (\text{curry}, \text{uncurry})$$

and the resulting type is $\text{LIST } \alpha \rightarrow \text{LIST } \beta \rightarrow \alpha \rightarrow \text{PAIR } (\beta \rightarrow \text{LIST } \alpha) (\beta \rightarrow \text{LIST } \beta)$. It is fairly straightforward to verify that $\tau_2^-(\tau_2^+(f)) = f$ so that the transformation $\tau_2^+(f)$ is deemed to be safe for all f . We now use the Eureka definition

$$\begin{aligned} \text{update}_2 \ xs \ ys \ x &= \tau_2^+(\text{update}_1) \ xs \ ys \ x \\ &= \text{topair } (\text{curry } \text{update}_1) \ xs \ ys \ x \end{aligned}$$

and it is fairly straightforward to get

$$\begin{aligned} \text{update}_2 \ \text{nil} \ \text{nil} \ x &\equiv \text{pair } g \ h \text{ WHERE } g \ y = \text{cons } x \ \text{nil} \\ &\quad \text{WHERE } h \ y = \text{cons } y \ \text{nil} \\ \text{update}_2 \ (\text{cons } x' \ xs) \ (\text{cons } y' \ ys) \ x \\ &\equiv \text{pair } g \ h \text{ WHERE } g \ y = \text{IF } \text{eq } x \ x' \text{ THEN } \text{cons } x \ xs \\ &\quad \text{ELSE CASE } \text{update}_2 \ xs \ ys \ x \text{ OF pair } g' \ h' \Rightarrow \text{cons } x' \ (g' \ y) \\ &\quad \text{WHERE } h \ y = \text{IF } \text{eq } x \ x' \text{ THEN } \text{cons } y \ ys \\ &\quad \text{ELSE CASE } \text{update}_2 \ xs \ ys \ x \text{ OF pair } g' \ h' \Rightarrow \text{cons } y' \ (h' \ y) \end{aligned}$$

In the *third stage* there is only one disagreement point in the type

$$\text{LIST } \alpha \rightarrow \text{LIST } \beta \rightarrow \alpha \rightarrow \text{PAIR } (\beta \rightarrow \text{LIST } \alpha) (\beta \rightarrow \text{LIST } \beta)$$

namely the root of the subtree $\beta \rightarrow \text{LIST } \alpha$. We shall here use Transformation 5 in Table 3 to replace the function type $\beta \rightarrow \text{LIST } \alpha$ with the list type $\text{LIST } \alpha$. Certainly, such a transformation will not be valid in all cases but if we are in a context in which we only consider constant functions from $\beta \rightarrow \text{LIST } \alpha$ then it will indeed be a valid transformation. The transformation we shall apply is

$$\tau_3 = \text{id} \rightarrow \text{id} \rightarrow \text{id} \rightarrow \text{PAIR } (\text{forget}, \text{recall}) \text{id}$$

and the resulting type is $\text{LIST } \alpha \rightarrow \text{LIST } \beta \rightarrow \alpha \rightarrow \text{PAIR } (\text{LIST } \alpha) (\beta \rightarrow \text{LIST } \beta)$. Using the Eureka definition

$$\begin{aligned} \text{update}_3 \text{ xs ys x} &= \tau_3^+(\text{update}_2) \text{ xs ys x} \\ &= \text{CASE update}_2 \text{ xs ys x OF pair g h} \Rightarrow \text{pair (forget g) h} \end{aligned}$$

we can use the fold/unfold framework and get

$$\begin{aligned} \text{update}_3 \text{ nil nil x} &\equiv \text{pair (cons x nil) h WHERE h y = cons y nil} \\ \text{update}_3(\text{cons x' xs}) (\text{cons y' ys}) x & \\ \equiv \text{pair (IF eq x x' THEN cons x xs} & \\ \text{ELSE CASE update}_3 \text{ xs ys x OF pair xs' h' } \Rightarrow \text{cons x' xs') h} & \\ \text{WHERE h y = IF eq x x' THEN cons y ys} & \\ \text{ELSE CASE update}_3 \text{ xs ys x OF pair xs' h' } \Rightarrow \text{cons y' (h' y)} & \end{aligned}$$

As indicated above it cannot be proved that $\tau_3^+(f)$ is deemed to be safe for all f but one may verify that $\tau_3^+(\text{update}_2)$ is indeed deemed to be safe.

In the *fourth stage* it is observed that the type

$$\text{LIST } \alpha \rightarrow \text{LIST } \beta \rightarrow \alpha \rightarrow \text{PAIR (LIST } \alpha) (\beta \rightarrow \text{LIST } \beta)$$

still has a disagreement point, namely the root of the subtype $\text{LIST } \beta \rightarrow \alpha \rightarrow \dots$. Transformation 3 of Table 2 can be used to swap the order of the second and third arguments so that the overall type becomes

$$\text{LIST } \alpha \rightarrow \alpha \rightarrow \text{LIST } \beta \rightarrow \text{PAIR (LIST } \alpha) (\beta \rightarrow \text{LIST } \beta)$$

The *fifth stage* takes care of the disagreement point corresponding to the subtype $\text{LIST } \beta \rightarrow \text{PAIR } \dots$ by applying Transformation 4 of Table 3. The overall type then becomes

$$\text{LIST } \alpha \rightarrow \alpha \rightarrow \text{PAIR (LIST } \beta \rightarrow \text{LIST } \alpha) (\text{LIST } \beta \rightarrow \beta \rightarrow \text{LIST } \beta)$$

The *sixth stage* is similar to stage three and will give a result of type

$$\text{LIST } \alpha \rightarrow \alpha \rightarrow \text{PAIR (LIST } \alpha) (\text{LIST } \beta \rightarrow \beta \rightarrow \text{LIST } \beta)$$

This type has no disagreement points and the transformation process stops. We shall not give the details of the fold/unfold steps of these transformations but only mention that the final function update_6 will turn out to be defined by

$$\begin{aligned} \text{update}_6 \text{ nil x} &= \text{pair (cons x nil) h WHERE h nil y = cons y nil} \\ \text{update}_6 (\text{cons x' xs}) x &= \text{pair (IF eq x x' THEN cons x xs} \\ &\quad \text{ELSE CASE update}_6 \text{ xs x OF pair xs' h' } \Rightarrow \text{cons x' xs') h} \\ \text{WHERE h (cons y' ys) y} &= \text{IF eq x x' THEN cons y ys} \\ &\quad \text{ELSE CASE update}_6 \text{ xs x OF pair xs' h' } \Rightarrow \text{cons y' (h' ys y)} \end{aligned}$$

5 Conclusion

The main point of the present paper was to construct Eureka definitions needed to apply the fold/unfold framework of Burstall and Darlington [1]. This involved systematic definitions of transformation functions so as to perform transformations in context and it involved a way of identifying the points, called disagreement points, where it would be sensible to perform a transformation. However, further work is needed concerning how to choose which disagreement point to work from (in case there are many) and to choose which transformation to apply (in case there are many candidates).

Our approach has been to let the *type of the function* guide the transformation process. This is contrary to the tupling, composition and generalization tactics of [2] where the *structure of the function definition* is used. Rather our work is related to the enhancement developed in [2] for semi-automatically generating directives to the transformation system. In particular the power of the AUTO feature [2] could be increased by using types to indicate what recursive calls to generate.

(This is already implicit in the AUTO feature but in a limited way.) The idea of using the types to guide the transformations can also be found in e.g. [1,11].

In this paper we used an (informal) notion of binding time to determine the disagreement points. A more thorough discussion of binding time analysis may be found in e.g. [9,5,7] and the transformation of programs based on binding time information can also be found in [10,7]. However, our approach is *not* ultimately connected with the use of binding time information. In [8] we show that an annotation based on *processor numbers* can be used to transform programs so as to obtain more parallelism. This is achieved by distributing shared data on individual processors.

Acknowledgement: The title of this paper was inspired by [14]. The work has been supported by the Danish Natural Science Research Council.

References

- [1] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 1977.
- [2] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 1982.
- [3] P. Hudak and P. Wadler. *Report on the Functional Programming Language Haskell*. Technical Report, Glasgow University, 1988.
- [4] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986.
- [5] J. Launchbury. Projections for specialization. In *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, North-Holland, 1988.
- [6] R. Milner. The standard ML core language. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, 1984.
- [7] T. Mogensen. Separating binding times in language specifications. In *Proc. of Functional Programming and Computer Architecture*, 1989.
- [8] F. Nielson and H. Riis Nielson. *Forced transformations of OCCAM programs*. Technical Report, Aarhus University, 1989.
- [9] H. Riis Nielson and F. Nielson. Automatic binding time analysis for a typed λ -calculus. *Science of Computer Programming*, 1988.
- [10] H. Riis Nielson and F. Nielson. Transformations on higher-order functions. In *Proc. of Functional Programming and Computer Architecture*, 1989.
- [11] W. Polak. *Compiler Specification and Verification*. LNCS 124, Springer, 1981.
- [12] J. C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings 2nd ICALP*, Springer LNCS 14, 1974.
- [13] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Language and Computer Architecture*, Springer LNCS 201, 1985.
- [14] P. Wadler. Theorems for free! In *Proc. 4th international conference on functional programming language and computer architecture*, ACM Press, 1989.