A Fold for All Seasons *

Tim Sheard   Leonidas Fegaras

Pacific Software Research Center
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999
{sheard,fegaras}@cse.ogi.edu

Abstract

Generic control operators, such as *fold*, can be generated from algebraic type definitions. The class of types to which these techniques are applicable is generalized to all algebraic types definable in languages such as Miranda and ML, i.e. mutually recursive sums-of-products with tuples and function types. Several other useful generic operators, also applicable to every type in this class, also are described.

A *normalization algorithm* which automatically calculates improvements to programs expressed in a language based upon folds is described. It reduces programs, expressed using fold as the exclusive control operator, to a canonical form. Based upon a generic *promotion theorem*, the algorithm is facilitated by the explicit structure of fold programs rather than using an analysis phase to search for implicit structure. Canonical programs are minimal in the sense that they contain the fewest number of fold operations. Because of this property, the normalization algorithm has important applications in program transformation, optimization, and theorem proving.

In addition a generic promotion theorem is identified for each of the other operators. It is *hoped* that these theorems can be the basis of normalization algorithms for the other operators as well.

1 Introduction

It has been argued that *functional languages are important because they support forms of modularity that are difficult to support in other languages* [16, 10]. Recent work by Malcolm [12], Meijer, Fokkinga, and Paterson [13], and Cockett with the programming language *Charity* [2] has suggested an even higher level of modularity and abstraction may be obtained by the use of generic control structures which capture patterns of recursion for a large class of types in a uniform way. This is important for several reasons:

- **Abstraction.** It allows the specification of algorithms independent of the type of data structures they are to operate on, since the control structure of the algorithm is generated for each datatype.
- **Genericity.** It allows the statement, proof, and use of generic theorems. In this manner, a theorem can be stated and proved in the abstract and then applied in many different concrete instances.
- **Structure.** Programming with recursive definitions and higher order functions is a powerful technique but it does not impose much structure on the form of the programs that may be expressed. This desire for structure is motivated for many of the same reasons that the "structured programming" methodology for imperative programs was motivated, but it has an even larger payback. While structure can facilitate understanding and maintainability in either paradigm, functional programs are often the target of transformation and optimization techniques. These techniques generally search for implicit structure in programs to validate particular transformations. If structure is explicit, rather than implicit, the job of the transformation system is made easier.

It is important that these operators be as widely applicable as possible and that transformation techniques exploiting these operators be developed, thus the contribution of this work is two fold:

- First, while these operators have previously been restricted to simple sums-of-products datatypes, they are in fact much more general. In this paper we extend these control structures to mutually recursive sums-of-products with tuples and function types.
- Second, rather than use general laws about these operators in an ad hoc way, we show that these laws can support *calculation-based* (rather than a search-based) program transformation methods. We introduce a *normalization algorithm* which automatically calculates improvements to programs expressed in a language based upon one of these operators. It reduces programs, (expressed using fold as the exclusive

* A version of this paper will appear in the Proceedings of the Conference on Functional Programming and Computer Architecture. Copenhagen June, 1993. Tim Sheard is supported in part by a grant from OACIS and Tektronix, Leonidas Fegaras is supported by contract DAAB07-91-C-Q518 (ARPA order number 018) from DARPA.

control operator), to a canonical form. *Canonical* programs are minimal in the sense that they contain the fewest number of fold operations.

The normalization algorithm exploits the explicit structure of fold programs. Therefore programs in this form can be transformed using fully automated tools. Because canonical programs are minimal, the normalization algorithm has important applications in program transformation, optimization, and theorem proving. For example:

- Automation of the unfold-simplify-fold method [4] without the intervention of explicit laws. In addition, "eureka" steps are not necessary here during folding to a function call as they are incorporated in the generalization part of the normalization algorithm.
- Automation of the techniques of listlessness, deforestation and fusion [15, 1, 8, 14] for a class of fold programs. These techniques remove unnecessary intermediate data structures produced during nested calls.
- Automation of a simple form of partial evaluation. If the parameter of a fold term is instantiated with other fold terms, (for example a piece of "data", i.e. a constant or ground term constructed wholly from constructors), then the normalization algorithm will reduce this program into canonical form that is specific to these instantiations and, furthermore, any "paths" in the original program not reachable with the current data will be removed from the resulting program.
- Theorem Proving. The equality of two programs can be checked by reducing them both to canonical form and comparing the results.

In Section 2 we review the work of Meijer, et al. [13]. We begin by introducing a different notation which originates from the work of Hagino [9] and is extended by Cockett [2]. We feel that this notation makes these ideas more accessible to functional programmers. We conclude this section by using this notation to formally define the familiar fold operator for simple algebraic types.

In Sections 3 and 4 we extend this notation to cover the complete set of types definable in languages such as Miranda and ML by extending it to cover mutually recursive sums-of-products with function types.

In Section 5 we define a term language in which fold is the only control structure, and introduce a reduction algorithm which we prove reduces fold programs to canonical form. One example of a reduction is described.

In Sections 6 – 7 we use our notation to define a number of other patterns of recursion in various levels of detail. Each pattern is accompanied by a generic promotion theorem.

In Section 8 we briefly compare our work to previous results.

In the last section we conclude and discuss future work.

2 Background

In this section we review previous work [12, 13] and introduce an alternative notation that will facilitate extension.

2.1 Simple Types

The type definitions considered in this section are the simple sums-of-product types defined by using recursive equations of the form:

$$\begin{array}{rcl} T(\alpha_1,\ldots,\alpha_p) & = & C_1(t_{1,1},\ldots,t_{1,m_1}) \\ & | & \cdots \\ & | & C_n(t_{n,1},\ldots,t_{n,m_n}) \end{array}$$

where $\alpha_1,\ldots,\alpha_p$ denote type variables, the C_i are names of value constructor functions, and $t_{i,j}$ are either type variables (in the set $\alpha_1,\ldots,\alpha_p$) or instantiations of sums-of-products types, including uniform[1] use of the type T itself. No mutually recursive types or types with tuples or function types are allowed. We will lift these restrictions in Sections 3 and 4

For example, the following are simple sums-of-products type definitions:

$$\begin{array}{rcl} \mathrm{boolean} & = & \mathrm{False} \mid \mathrm{True} \\ \mathrm{prod}(\alpha,\beta) & = & \mathrm{Pair}(\alpha,\beta) \\ \mathrm{list}(\alpha) & = & \mathrm{Nil} \mid \mathrm{Cons}(\alpha,\mathrm{list}(\alpha)) \\ \mathrm{int} & = & \mathrm{Zero} \mid \mathrm{Succ}(\mathrm{int}) \\ \mathrm{tree}(\alpha) & = & \mathrm{Tip}(\alpha) \mid \mathrm{Node}(\mathrm{tree}(\alpha),\mathrm{tree}(\alpha)) \\ \mathrm{bush}(\alpha) & = & \mathrm{Leaf}(\alpha) \mid \mathrm{Branch}(\mathrm{list}(\mathrm{bush}(\alpha))) \end{array}$$

Functions manipulating values of these types will use a pattern of recursion related to the pattern of recursion in the type definitions. To capture these patterns we turn to the categorical notion of a functor.

2.2 Functors

We define below the functor, E, (which is really the morphism part of a categorical functor). The property of any functor, F, that we require is:

$$F(f_1,\ldots,f_n)\circ F(h_1,\ldots,h_n) \;=\; F(f_1\circ h_1,\ldots,f_n\circ h_n)$$

which may easily be verified for our definition below.

Definition 1 (The Functor E) *Associated with each constructor, $C_i : (t_{i,1},\ldots,t_{i,m_i}) \to T(\alpha_1,\ldots,\alpha_p)$ is a $(p+1)$-adic functor[2]:*

$$\begin{array}{l} E_i^T(f_1,\ldots,f_p,f_T) \;= \\ \qquad \lambda(x_{i,1},\ldots,x_{i,m_i}).(K[t_{i,1}]x_{i,1},\ldots,K[t_{i,m_i}]x_{i,m_i}) \end{array}$$

where the bound variables, $x_{i,j}$ have type, $t_{i,j}$, and $K[t_{i,j}]$ represents a function that can be obtained by the following rules:

$$\begin{array}{rcll} K[\alpha_k] & = & f_k & (1) \\ K[T(\alpha_1,\ldots,\alpha_p)] & = & f_T & (2) \\ K[S(t_1,\ldots,t_q)] & = & \mathrm{map}^S\,(K[t_1],\ldots,K[t_q]) & (3) \end{array}$$

where map^S, for a previously defined type $S(\alpha_1,\ldots,\alpha_q)$, can be defined using the functors, E_i^S:

$$(\mathrm{map}^S(f_1,\ldots,f_q))\circ C_i \;=\; C_i\circ(E_i^S(f_1,\ldots,f_q,\mathrm{map}^S(f_1,\ldots,f_q)))$$

[1] By *uniform* we mean that recursive use of the type constructor T is restricted to applications to the free type variables in the same order as they appear on the left hand side of the recursive type equation.

[2] Where p is the number of universally quantified type variables in the left hand side of T's type equation.

For example, for type *list*:

$$\begin{array}{rcl} E_{Nil}(f,g) & = & \lambda().() \\ E_{Cons}(f,g) & = & \lambda(x,y).(f\,x,\ g\,y) \end{array}$$

and for *bush*:

$$\begin{array}{rcl} E_{Leaf}(f,g) & = & \lambda(x).(f\,x) \\ E_{Branch}(f,g) & = & \lambda(x).(\mathrm{map}^{list}\ (g)\ x) \end{array}$$

where

$$\begin{array}{lcl} \mathrm{map}^{list}\ (f)\ \mathrm{Nil} & = & \mathrm{Nil} \\ \mathrm{map}^{list}\ (f)\ (\mathrm{Cons}(a,l)) & = & \mathrm{Cons}(f\ a, \mathrm{map}^{list}\ (f)\ l) \end{array}$$

2.3 Folds

With this notation it is now possible to describe the familiar *fold* (catamorphism [13]) operator for any simple sums-of-products type.

Definition 2 (Fold) *The fold function over $T(\alpha_1 \dots \alpha_p)$ is defined by the following set of recursive equations, one for each constructor, C_i:*

$$(\mathrm{fold}^T(\overline{f})) \circ C_i = f_i \circ (E_i^T(id_1, \dots, id_p, \mathrm{fold}^T(\overline{f})))$$

where $\overline{f} = (f_1, \dots, f_n)$ and for each index j, id_j is the identity function.

We call each f_i in $\overline{f}$ an *accumulating function.*

For example, the list fold is defined as:

$$\begin{array}{l} \mathrm{fold}^{list}\ (f_n, f_c)\ \mathrm{Nil}\ =\ f_n() \\ \mathrm{fold}^{list}\ (f_n, f_c)\ (\mathrm{Cons}(a,l))\ =\ f_c(a, \mathrm{fold}^{list}\ (f_n, f_c)\ l) \end{array}$$

The following are some examples of computations that can be defined using folds:

$$\begin{array}{lcl} \mathrm{copy}(x) & = & \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{Cons}(a,r))\ x \\ \mathrm{append}(x,y) & = & \mathrm{fold}^{list}(\lambda().y, \lambda(a,r).\mathrm{Cons}(a,r))\ x \\ \mathrm{length}(x) & = & \mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\ x \\ x+y & = & \mathrm{fold}^{int}(\lambda().y, \lambda(r).\mathrm{Succ}(r))\ x \\ x \times y & = & \mathrm{fold}^{int}(\lambda().\mathrm{Zero}, \lambda(r).y+r)\ x \\ x \wedge y & = & \mathrm{fold}^{boolean}(\lambda().\mathrm{False}, \lambda().y)\ x \\ \text{if } x \text{ then } y \text{ else } z & = & \mathrm{fold}^{boolean}(\lambda().z, \lambda().y)\ x \end{array}$$

The general law which applies to all fold functions is called the *promotion theorem.* We will use this theorem as a major component of our automated transformation algorithm, called the *normalization algorithm.* The promotion theorem states that the composition of any function g with some fold is another fold whose accumulating functions are related to the accumulating functions of the original fold by fixed equations. The normalization algorithm will describe how these new accumulating functions can be calculated. The promotion theorem for folds has appeared in the literature using various notations [12, 13]:

Theorem 1 (The Fold Promotion Theorem)

$$\frac{\forall i:\ \phi_i \circ E_i^T(id_1, \dots, id_p, g)\ =\ g \circ f_i}{g \circ \mathrm{fold}^T(\overline{f})\ =\ \mathrm{fold}^T(\overline{\phi})}$$

Proof: Let $\eta = g \circ \mathrm{fold}^T(\overline{f})$ and C_i a constructor of T. Then

$$\begin{array}{l} \eta \circ C_i = g \circ \mathrm{fold}^T(\overline{f}) \circ C_i \\ \quad = g \circ f_i \circ E_i^T(id_1, \dots, id_p, \mathrm{fold}^T(\overline{f})) \\ \qquad\qquad \text{by Definition 2} \\ \quad = \phi_i \circ E_i^T(id_1, \dots, id_p, g) \circ E_i^T(id_1, \dots, id_p, \mathrm{fold}^T(\overline{f})) \\ \qquad\qquad \text{by premise} \\ \quad = \phi_i \circ E_i^T(id_1, \dots, id_p, g \circ \mathrm{fold}^T(\overline{f})) \\ \qquad\qquad \text{by composition under } E_i \\ \quad = \phi_i \circ E_i^T(id_1, \dots, id_p, \eta) \\ \qquad\qquad \text{by definition of } \eta \end{array}$$

Therefore, $\eta \circ C_i\ =\ \phi_i \circ E_i^T(id_1, \dots, id_p, \eta)$, which shows that η has a form of a fold with accumulating functions $\overline{\phi}$ (by Definition 2). Thus $\eta = \mathrm{fold}^T(\overline{\phi})$. □

For example the promotion theorem for *list* is:

$$\frac{\begin{array}{lcl} \phi_n() & = & g(f_n()) \\ \phi_c(a, g(r)) & = & g(f_c(a,r)) \end{array}}{g(\mathrm{fold}^{list}(f_n, f_c)\ x)\ =\ \mathrm{fold}^{list}(\phi_n, \phi_c)\ x}$$

3 Types with Tuples and Exponentials

A tuple type, T, is a sums-of-products type where the products may contain tuple types, $t_1 \times \dots \times t_n$. For example:

$$\mathrm{association}(\alpha, \beta)\quad =\quad \mathrm{Alist}(\mathrm{list}(\alpha \times \beta))$$

An exponential type, T, is a sums-of-products type where the products may contain function types, $\beta \rightarrow \gamma$. For example, potentially infinite lists can be defined in a strict language as an exponential type as follows:

$$\mathrm{ilist}(\alpha)\quad =\quad \mathrm{Inil}\ \mid\ \mathrm{Icons}(\alpha, () \rightarrow \mathrm{ilist}(\alpha))$$

where () is the unit type (the unique type of the empty tuple).

To define a fold over sums-of-products with tuples and function types we need only to generalize the functor, E. We do this by adding the following rules to K in Definition 1.

$$K[t_1, t_2]\ =\ \lambda(x_1, x_2).\,(K[t_1]\,x_1, K[t_2]\,x_2) \tag{4}$$

$$K[u \rightarrow v]\ =\ \lambda h.\ K[v] \circ h \circ K[u] \tag{5}$$

$$K[()]\ =\ id \tag{6}$$

For example, for type *ilist*:

$$\begin{array}{lcl} E_{Inil}(f_\alpha, f_T) & = & \lambda().() \\ E_{Icons}(f_\alpha, f_T) & = & \lambda(x,g).(f_\alpha\,x,\ (\lambda h.\ f_T \circ h \circ id)\,g) \\ & = & \lambda(x,g).(f_\alpha\,x,\ f_T \circ g) \end{array}$$

While the fold function derived from this generalized functor is defined for all sums-of-products with tuples and function types it is not particularly useful. To be useful, it is necessary to restrict each function type to be covariant in T [5], otherwise $K[t]$ may not be a functor.

For example the fold for *ilist* is:

$$\begin{array}{lcl} \mathrm{fold}^{ilist}\ (f_n, f_c)\ \mathrm{Inil} & = & f_n() \\ \mathrm{fold}^{ilist}\ (f_n, f_c)\ \mathrm{Icons}(a,g) & = & f_c(a, (\mathrm{fold}^{ilist}\ (f_n, f_c)) \circ g) \end{array}$$

4 Mutually Recursive Types

A set of mutually recursive types can be defined by a set of mutually recursive equations of the form:

$$\begin{array}{lcl} \bigwedge_{s \in [1 \dots r]}\quad T_s(\alpha_1, \dots, \alpha_p) & = & C_1^{T_s}(t_{1,1}, \dots, t_{1,m_1}) \\ & \mid & \dots \\ & \mid & C_n^{T_s}(t_{n,1}, \dots, t_{n,m_n}) \end{array}$$

where the $T_1, \ldots, T_r$ are the types being defined. For example consider the type which models a very simple expression language with local declarations[3]:

$$\begin{array}{lrcl} & \exp(\alpha,\beta) & = & \mathrm{Var}(\alpha) \\ & & | & \mathrm{Let}(\mathrm{dec}(\alpha,\beta), \exp(\alpha,\beta)) \\ & & | & \mathrm{Apply}(\exp(\alpha,\beta), \exp(\alpha,\beta)) \\ \bigwedge & \mathrm{dec}(\alpha,\beta) & = & \mathrm{Val}(\beta, \exp(\alpha,\beta)) \\ & & | & \mathrm{Fun}(\mathrm{string}, \beta, \exp(\alpha,\beta)) \end{array}$$

As in the simple case, associated with each constructor, C_i, is functor, E_i^T. If there are r mutually recursive types, $T_1, \ldots, T_r$, and constructor $C_i^{T_s} : (t_{i,1}, \ldots, t_{i,m_i}) \rightarrow T_s$ then $E_i^{T_s}$ is a $(p+r)$-adic functor:

$$E_i^{T_s}(f_1, \ldots, f_p, f_{T_1}, \ldots, f_{T_r}) = \\ \lambda(x_{i,1}, \ldots, x_{i,m_i}).(K[t_{i,1}]x_{i,1}, \ldots, K[t_{i,m_i}]x_{i,m_i})$$

where the bound variables, $x_{i,j}$ have type, $t_{i,j}$, we replace rule (2) in the defintion of K in Definition 1 with:

$$K[T_i(\alpha_1, \ldots, \alpha_p)] \;=\; f_{T_i} \qquad (2)$$

Generic functions, such as map and fold, defined over mutually recursive types will be mutually recursive functions. For example, the map functions for a set of mutually recursive types, $T_1, \ldots, T_r$, are defined by the equations:

$$\bigwedge_{s \in [1 \ldots r]} \mathrm{map}^{T_s}(\overline{f}) \circ C_i^{T_s} = \\ C_i^{T_s} \circ E_i^{T_s}(f_1, \ldots, f_p, \mathrm{map}^{T_1}(\overline{f}), \ldots, \mathrm{map}^{T_r}(\overline{f}))$$

where $(\overline{f})$ is $(f_1, \ldots, f_p)$.

For example, the *map* for *exp* and *dec* is given by:

$$\begin{array}{lcl} \mathrm{map}^{exp}(\overline{h})(\mathrm{Var}\, x) & = & \mathrm{Var}(f\, x) \\ \mathrm{map}^{exp}(\overline{h})(\mathrm{Let}(x,y)) & = & \mathrm{Let}(\mathrm{map}^{dec}(\overline{h})\, x, \mathrm{map}^{exp}(\overline{h})\, y) \\ \mathrm{map}^{exp}(\overline{h})(\mathrm{Apply}(x,y)) & = & \mathrm{Apply}(\mathrm{map}^{exp}(\overline{h})\, x, \mathrm{map}^{exp}(\overline{h})\, y) \\ \bigwedge & & \\ \mathrm{map}^{dec}(\overline{h})(\mathrm{Val}(x,y)) & = & \mathrm{Val}(g\, x, \mathrm{map}^{exp}(\overline{h})\, y) \\ \mathrm{map}^{dec}(\overline{h})(\mathrm{Fun}(x,y,z)) & = & \mathrm{Fun}(x, y, \mathrm{map}^{exp}(\overline{h})\, z) \end{array}$$

where $(\overline{h})$ is $((f),(g))$.

Definition 3 (Mutually Recursive Fold) *The mutually recursive fold functions over the types $T_1, \ldots, T_r$ is defined by the following set of mutually recursive equations:*

$$\bigwedge_{s \in [1 \ldots r]} \mathrm{fold}^{T_s}(\overline{f}) \circ C_i^{T_s} = \\ f_i^{T_s} \circ E_i^{T_s}(id_1, \ldots, id_p, \mathrm{fold}^{T_1}(\overline{f}), \ldots, \mathrm{fold}^{T_r}(\overline{f}))$$

Where $(\overline{f}) = ((f_1^{T_1}, \ldots, f_{n_1}^{T_1}), \ldots, (f_1^{T_r}, \ldots, f_{n_r}^{T_r}))$, a tuple of tuples of accumulating functions.

For example, the mutually recursive fold functions for *exp* and *dec* are:

$$\begin{array}{lcl} \mathrm{fold}^{exp}(\overline{f})(\mathrm{Var}\, x) & = & Vf\, x \\ \mathrm{fold}^{exp}(\overline{f})(\mathrm{Let}(x,y)) & = & Lf(\mathrm{fold}^{dec}(\overline{f})\, x, \mathrm{fold}^{exp}(\overline{f})\, y) \\ \mathrm{fold}^{exp}(\overline{f})(\mathrm{Apply}(x,y)) & = & Af(\mathrm{fold}^{exp}(\overline{f})\, x, \mathrm{fold}^{exp}(\overline{f})\, y) \\ \bigwedge & & \\ \mathrm{fold}^{dec}(\overline{f})(\mathrm{Val}(x,y)) & = & Valf(x, \mathrm{fold}^{exp}(\overline{f})\, y) \\ \mathrm{fold}^{dec}(\overline{f})(\mathrm{Fun}(x,y,z)) & = & Funf(x, y, \mathrm{fold}^{exp}(\overline{f})\, z) \end{array}$$

where $(\overline{f})$ is $((Vf, Lf, Af), (Valf, Funf))$.

[3] As one might find in an ML like language with only variables, application, and let expressions.

Theorem 2 (Mutually Recursive Fold Promotion)

$$\frac{\forall f_i \in (\overline{f})_k : \; \phi_i \circ E_i^{T_k}(id_1, \ldots, id_p, id_1 \ldots, g, \ldots, id_r) \;=\; g \circ f_i}{g \circ \mathrm{fold}^{T_k}(\overline{f}_1, \ldots, \overline{f}_k, \ldots, \overline{f}_r) \;=\; \mathrm{fold}^{T_k}(\overline{f}_1, \ldots, \overline{\phi}, \ldots, \overline{f}_r)}$$

Proof: Analogous to Theorem 1. □

That is, for a fold over the type T_k, the k^{th} type in a set of mutually recursive types, $(T_1, \ldots, T_r)$, the function, g, is placed in only the k^{th} recursive position of E_i.

5 Normalization of Fold Expressions

The normalization algorithm reduces any fold applied to another fold into a fold applied to a variable. This reduced program has fewer folds. Since every fold builds a data structure, improved programs will build fewer intermediate data structures.

Program normalization is accomplished by pushing the outer fold into the accumulating functions of the inner fold, as is directed by the promotion theorem. This is a generalization of nested loop fusion to arbitrary types because the outer fold will be pushed inside the inner one until it is eliminated by the generalizations introduced by the normalization algorithm.

5.1 The Term Language

In the following definitions we assume that programs are well-typed by a ML-style polymorphic type-checker, and that all folds are over mutually recursive sums-of-products types that may include tuples and function types. Any non-mutually recursive fold is handled as a fold over mutually recursive with only a single type.

Definition 4 (The Term Language) *A* program *in the term language has the form $\lambda(x_1, \ldots, x_n).t$, where each x_i is a variable and t is a term. Each term has one of the following forms:*

- variable: *x, bound in some outer lambda abstraction;*
- construction: *$C(t_1, \ldots, t_n)$, where C is a constructor and each t_i is a term;*
- fold: *$\mathrm{fold}^{T_s}((f_1^{T_1}, \ldots, f_{n_1}^{T_1}), \ldots, (f_1^{T_r}, \ldots, f_{n_r}^{T_r}))\, t$, that is, a mutually recursive fold over a term, t, of type T_s, where T_s is one of a set of mutually recursive types $T_1, \ldots, T_r$. Each $f_i^{T_k}$ has the form $\lambda(x_1, \ldots, x_m).t_i$, where each x_j is a variable and t_i is a term.*

Definition 5 (Accumulative Result Variable) *Each accumulating function $f_i^{T_k}$ in a fold in the term language has the form $\lambda(x_1, \ldots, x_m).t$, where the types of the bound variables, $x_1, \ldots, x_m$, are associated with the types, $t_1, \ldots, t_m$, in the domain of the corresponding constructor $C_i^{T_k}$. Each bound variable, x_j, whose associated type, t_j, mentions at least one of the recursive types $T_1, \ldots, T_r$ is an* accumulative result variable*.*

For example, in the definition of append

$$\mathrm{append}(x,y) \;=\; \mathrm{fold}^{list}(\lambda().y, \lambda(a,r).\mathrm{Cons}(a,r))\, x$$

r is an accumulative result variable.

Definition 6 (Potentially Normalizable Program) *A program in the term language is* potentially normalizable *if it does not contain terms* $\mathrm{fold}^T(\overline{f})\,t$*, where* t *contains a reference to an* accumulative result variable *which is bound in an outer lambda abstraction.*

For example of a term which is *not* potentially normalizable consider the reverse function normally defined by the recursion equations:

$$\begin{array}{lcl} \mathrm{rev\ Nil} & = & \mathrm{Nil} \\ \mathrm{rev}\ (\mathrm{Cons}(x, xs)) & = & \mathrm{append}(\mathrm{rev}\ xs,\ \mathrm{Cons}(x, \mathrm{Nil})) \end{array}$$

which can be expressed as the fold:

$$\begin{array}{l} \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{append}(r, \mathrm{Cons}(a, \mathrm{Nil})))\ x \\ \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \\ \qquad \lambda(a,r).\mathrm{fold}^{list}(\lambda().\mathrm{Cons}(a,\mathrm{Nil}), \lambda(b,s).\mathrm{Cons}(b,s))\ r)\ x \end{array}$$

where the *append* function has been expanded to a fold in the second equation. Any term using *rev* is not potentially normalizable, because the inner fold is over r, an accumulative result variable.

Definition 7 (Canonical Terms) *A canonical term is a term in which*

- *all folds in the term are over variables;*
- *none of these variables are accumulative result variables.*

Note that a canonical term is always a potentially normalizable term. We will show that if a program in our term language is potentially normalizable then it can be transformed into a canonical program by the normalization algorithm.

5.2 The Normalization Algorithm

The normalization algorithm is a meaning preserving transformation from term to term. It uses a parameter, ρ, which is a partial function from terms to variables. In our notation, $\rho[g/r]$ extends ρ with the mapping from g to r, while $\rho[(g_1,\ldots,g_n)/(r_1,\ldots,r_n)]$ extends ρ with the mappings from g_i to r_i.

The normalization algorithm consists of the following parts:

- *Generalization:* If the normalization algorithm derives a term mapped in ρ to some variable v, then this term is replaced by v.
- *Application to a Construction:* From the fold definition:

  $$\begin{array}{l} \mathrm{fold}^{T_s}(\overline{f})(C_i^{T_s}(\overline{u})) \ = \\ \qquad f_i^{T_s}(E_i^{T_s}(id_1,\ldots,id_p,\mathrm{fold}^{T_1}(\overline{f}),\ldots,\mathrm{fold}^{T_1}(\overline{f}))(\overline{u})) \end{array}$$

  E.g., for length $=$ $\mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))$:

  $$\mathrm{length}(\mathrm{Cons}(a,l)) \ = \ \mathrm{Succ}(\mathrm{length}(l))$$

- *Fold Promotion:* If the term is a composition

  $$g(\mathrm{fold}^{T_s}(\overline{f})\,x)$$

  where g is a fold, then the fold promotion theorem is applied to derive the term, $\mathrm{fold}^{T_s}(\overline{\phi})\,x$, where $\overline{\phi} = ((\phi_1^{T_1},\ldots,\phi_{n_1}^{T_1}),\ldots,(\phi_1^{T_r},\ldots,\phi_{n_r}^{T_r}))$ and for all k and i, $\phi_i^{T_k}$ is computed by recursively improving the equation:

  $$\phi_i^{T_k}(r_1,\ldots,r_{m_i}) \ = \ g(f_i^{T_k}(x_1,\ldots,x_{m_i}))$$

  where all x_i and r_i are new variables. In each case, ρ is extended with the mappings from terms in $(E_i^{T_k}(id_1,\ldots,id_q,id_1,\ldots,g_k,\ldots,id_r)(x_1,\ldots,x_{m_i}))$ to variables $r_1,\ldots,r_{m_i}$.

  For example, from the fold promotion theorem for lists:

  $$g(\mathrm{fold}^{list}(f_n,f_c)\,x) \ = \ \mathrm{fold}^{list}(\phi_n,\phi_c)\,x$$

  where

  $$\begin{array}{lcll} \phi_n() & = & g(f_n()) & \\ \phi_c(r_1,r_2) & = & g(f_c(x_1,x_2)) & \rho[x_1/r_1,\ g(x_2)/r_2] \end{array}$$

If C_i has type $(t_1,\ldots,t_m)\to T$, then ϕ_i satisfies $\phi_i(K[t_1]\,x_1,\ldots,K[t_m]\,x_m) = g(f_i(x_1,\ldots,x_m))$. In order to solve this equation for ϕ_i, we need to transform $g(f_i(x_1,\ldots,x_m))$ into a term $\mathcal{X}(K[t_1]\,x_1,\ldots,\ K[t_m]\,x_m)$, where $\mathcal{X}$ does not depend on any of x_i. Then generalizing $K[t_i]\,x_i$ to the variable r_i we find $\phi_i = \mathcal{X}$. The generalization function, ρ, plays this role.

The normalization algorithm can be implemented very efficiently. Whenever we apply the promotion theorem we annotate each function g (the left part of the application) with a new number. Then, instead of inserting $g(x_i)/r_i$ in ρ, for some variables x_i and r_i, we insert the triple of this number with x_i and r_i. Then the generalization phase is done by checking if the current term $f(x)$ was annotated by a number already in ρ. That way, the complexity of the normalization algorithm is linear in the size of the resulting canonical term.

5.3 One Example of Program Normalization

The following normalizes the composition length(append(x,y)) where:

$$\begin{array}{lcl} \mathrm{length}(x) & = & \mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\,x \\ \mathrm{append}(x,y) & = & \mathrm{fold}^{list}(f_n,f_c)\,x \\ & & \text{where} \left\{ \begin{array}{lcl} f_n & = & \lambda().y \\ f_c & = & \lambda(a,r).\mathrm{Cons}(a,r) \end{array} \right. \end{array}$$

We need to find some $\mathrm{fold}^{list}(\phi_n,\phi_c)\,x$ equivalent to the composition length(append(x,y)). The normalization algorithm for $g =$ length gives:

$$\begin{array}{lcl} \phi_n() & = & \mathrm{length}(f_n()) \\ & & \qquad \text{where } \rho = [\,] \\ & = & \mathrm{length}(y) \\ & = & \mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\,y \\ \phi_c(r_1,r_2) & = & \mathrm{length}(f_c(x_1,x_2)) \\ & & \qquad \text{where } \rho = [x_1/r_1,\ \mathrm{length}(x_2)/r_2] \\ & = & \mathrm{length}(\mathrm{Cons}(x_1,x_2)) \\ & = & \mathrm{Succ}(\mathrm{length}(x_2)) \\ & = & \mathrm{Succ}(r_2) \\ & & \qquad (\mathrm{length}(x_2) \text{ was generalized to } r_2) \end{array}$$

Therefore, the composition length(append(x,y)) is:

$$\begin{array}{l} \mathrm{fold}^{list}(\lambda().\mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\,y, \\ \qquad \lambda(r_1,r_2).\mathrm{Succ}(r_2))\,x \end{array}$$

Note that the intermediate list structure append(x,y) is no longer produced.

5.4 Correctness of the Normalization Algorithm

In this section we outline a proof that the normalization algorithm transforms any potentially normalizable term into canonical form. Here we consider folds consisting of sums-of-products types where the recursive type, T, does not appear as a parameter to some other parametric type. That is, we do not consider types such as $\text{bush}(\alpha) = \text{Leaf}(\alpha) \mid \text{Branch}(\text{list}(\text{bush}(\alpha)))$.

We can see from the definitions of the term language and canonical forms that folds, g, of the form $\text{fold}^T(\overline{f})\, t$ need to be rewritten into folds over variables.

If t is a construction, then we apply the *application-to-a-construction* rule. In that case the fold, g, is pushed inside to only those components of the construction that have recursive type.

If t is a fold, then g is pushed into the accumulators of the inner fold, as is directed by the fold promotion theorem, and some new mappings are attached to ρ.

After recursively applying the normalization algorithm only terms of the form $g(x)$, where x is a variable, remain. If x is an accumulative result variable, then, from the way ρ was extended during promotion, $g(x)$ is always bound in ρ to some new variable, r. This is the generalization phase of the algorithm. If x is not an accumulative result variable, then $g(x)$ remains as is. Therefore, all folds in the resulting term are over non-accumulative result variables. It is necessary to prove that after fold promotion there remain no references of the old accumulative result variables x_i in the improved $g(f_i(x_1, \ldots, x_{m_i}))$ (other than those of the form, $g(x_i)$, which have been generalized to r_i), leaving the term $\phi_i(r_1, \ldots, r_{m_i})$. There are two places that this could happen (that will be ruled out):

- An accumulative result variable, x, is inside a non-recursive subcomponent of a construction (because g is not pushed inside these components). This is impossible as it is ruled out by type considerations. Consider the term, $h(\text{Cons}(x, l))$, where x is an accumulative result variable. Either h is the identity, a construction or a fold. A fold is ruled out since this would imply a fold over a term containing an accumulative result variable. Note that the terms, $h(\text{Cons}(x, l))$ and x, must have the same type, $T(\alpha)$. This rules out the identity function and constructions since they lead to the circular type equations of the form $\text{list}(T(\alpha)) = T(\alpha)$, and all terms are well typed.
- An accumulative result variable appears in a term, t, traversed by a fold: $\text{fold}^T(\overline{f})\, t$ (because g is not pushed inside t during promotion). This too is ruled out given that all terms are potentially normalizable.

The composition of two or more canonical terms is a potentially normalizable term and, therefore, it can be reduced to canonical form. This makes our canonical term language closed under composition.

6 Other Patterns of Recursion

In this section we catalog other patterns of recursion which have appeared in the literature or been investigated by the authors.

6.1 Derivations

The Derive operator (anamorphism [13], unfold [3]) is the opposite of the fold operator. Rather than traversing an object of type T to produce a result value, it constructs an object of type T from a seed value. To describe derivations some additional mechanism is needed.

6.1.1 Prime Types

Associated with each type, T, is a related (non-recursive) type, T', such that if T is defined by:

$$T(\overline{\alpha}) \;=\; C_1(t_{1,1}, \ldots, t_{1,m_1}) \mid \cdots \mid C_n(t_{n,1}, \ldots, t_{n,m_n})$$

then T' is defined by:

$$T(\overline{\alpha}, R) \;=\; C_1'(s_{1,1}, \ldots, s_{1,m_1}) \mid \cdots \mid C_n'(s_{n,1}, \ldots, s_{n,m_n})$$

where $s_{i,j} = t_{i,j}[T(\alpha_1, \ldots, \alpha_p)/R]$. T' has one more universally quantified type variable. To obtain the right hand side of the equation defining T', this variable replaces all (recursive) occurrences of $T(\alpha_1, \ldots, \alpha_p)$ in the right hand side of the equation for T.

In category theory, T is often defined in terms of T', as its least fixed point. Since functional programmers are accustomed to writing recursive equations, we choose to define T' in terms of T. Some examples are:

$$\begin{aligned}
\text{list}(\alpha) &= \text{Nil} \mid \text{Cons}(\alpha, \text{list}(\alpha)) \\
\text{list}'(\alpha, R) &= \text{Nil}' \mid \text{Cons}'(\alpha, R) \\[1ex]
\text{bush}(\alpha) &= \text{Leaf}(\alpha) \mid \text{Branch}(\text{list}(\text{bush}(\alpha))) \\
\text{bush}'(\alpha, R) &= \text{Leaf}'(\alpha) \mid \text{Branch}'(\text{list}(R))
\end{aligned}$$

Note that the $\text{map}^{T'}$ defined by the equations:

$$\text{map}^{T'}(f_1, \ldots, f_p, f_R) \circ C_i' \;=\; C_i' \circ E_i^T(f_1, \ldots, f_p, f_R)$$

is a functor. Note that since T' is not recursive, $\text{map}^{T'}$ is not recursive either. And since T and T' have the same "shape", it is possible to use the functor E_i^T in the definition of $\text{map}^{T'}$. For example:

$$\begin{aligned}
\text{map}^{list'}\,(f, R)\ \text{Nil}' &= \text{Nil}' \\
\text{map}^{list'}\,(f, R)\ (\text{Cons}'(x, xs)) &= \text{Cons}'(f\ x, R\ xs)
\end{aligned}$$

The function in^T defined by $\text{in}^T = \text{fold}^{T'}(C_1, \ldots, C_n)$ can also be given by the equations:

$$\text{in}^T \circ C_i' \;=\; C_i$$

It injects a value of type $T'(\alpha_1, \ldots, \alpha_p, T(\alpha_1, \ldots, \alpha_p))$ to the type $T(\alpha_1, \ldots, \alpha_p)$. For example:

$$\begin{aligned}
\text{in}^{list}\ \text{Nil}' &= \text{Nil} \\
\text{in}^{list}\ (\text{Cons}'(x, xs)) &= \text{Cons}(x, xs)
\end{aligned}$$

Definition 8 (Derive) *The derive function with type $\beta \to T(\alpha_1, \ldots, \alpha_p)$ is given by a template of the form*

$$\text{Derive}^T\ P \;=\; \text{in}^T \circ (\text{map}^{T'}(id_1, \ldots, id_p, \text{Derive}^T\ P)) \circ P$$

The function P is called a splitting function *and has type $\beta \to T'(\alpha_1, \ldots, \alpha_p, \beta)$.*

For example, the derivation for *list* could be written as:

$$
\begin{aligned}
&\text{Derive}^{list}\ P\ x\ =\\
&\quad\text{case } P\ x \text{ of}\\
&\qquad \text{Nil}' \qquad\qquad \Rightarrow\ \text{Nil}\\
&\quad\mid\ \text{Cons}'(x, xs)\ \Rightarrow\ \text{Cons}(x, \text{Derive}^{list}\ P\ xs)
\end{aligned}
$$

Using this definition, the list of integers from starting point n, upto m, is given by the derivation:

$$\text{Derive}^{list}(\lambda x.\ \text{if } x \geq m \text{ then Nil}' \text{ else Cons}'(x, x+1))\ n$$

Unlike folds, which always terminate (if their accumulating functions terminate), it is possible to construct derivations which construct infinite values. For example, the infinite list from starting point n, is given by the derivation:

$$\text{Derive}^{list}(\lambda x.\ \text{Cons}'(x, x+1))\ n$$

In this example the splitting function never returns Nil', so the derivation expands indefinitely.

Theorem 3 (The Derive Promotion Theorem)

$$\frac{\text{map}^{T'}(id_1, \ldots, id_p, g) \circ \phi\ =\ P \circ g}{(\text{Derive}^T\ P) \circ g\ =\ \text{Derive}^T\ \phi}$$

Proof: Let $\eta = (\text{Derive}^T\ P) \circ g$.

$$
\begin{aligned}
\eta &= (\text{Derive}^T\ P) \circ g\\
&= \text{in}^T \circ \text{map}^{T'}(id_1, \ldots, id_p, \text{Derive}^T\ P) \circ P \circ g\\
&\qquad \text{by Definition 8}\\
&= \text{in}^T \circ \text{map}^{T'}(id_1, \ldots, id_p, \text{Derive}^T\ P)\\
&\qquad \circ\ \text{map}^{T'}(id_1, \ldots, id_p, g) \circ \phi\\
&\qquad \text{by premise}\\
&= \text{in}^T \circ \text{map}^{T'}(id_1, \ldots, id_p, (\text{Derive}^T\ P) \circ g) \circ \phi\\
&\qquad \text{by composition under map}^{T'}\\
&= \text{in}^T \circ \text{map}^{T'}(id_1, \ldots, id_p, \eta) \circ \phi\\
&\qquad \text{by definition of } \eta.
\end{aligned}
$$

Thus we see that η has the form of a derivation with splitting function ϕ. □

6.2 Primitive Recursion

While the fold function for a type T is a useful function it is not as general as one would like. Consider the factorial function and the fold for *int*:

$$
\begin{aligned}
\text{fold}^{int}\ (f_z, f_s)\ \text{Zero} &= f_z()\\
\text{fold}^{int}\ (f_z, f_s)\ \text{Succ}(x) &= f_s(\text{fold}^{int}\ (f_z, f_s)\ x)
\end{aligned}
$$

Straight-forward[4] attempts to capture factorial as a fold over the natural numbers fail since the accumulating function f_s is not passed a copy of the value of x, but only the result of recursively transforming x. A fold-like function in which the accumulating functions are passed both the original variable, and the recursively transformed value can capture (in a straight-forward manner) a larger class of functions. Such a function for *int*, and a definition for factorial in terms of it follows:

$$
\begin{aligned}
\text{Pr}^{int}\ (f_z, f_s)\ \text{Zero} &= f_z()\\
\text{Pr}^{int}\ (f_z, f_s)\ \text{Succ}(x) &= f_s(x, \text{Pr}^{int}\ (f_z, f_s)\ x)
\end{aligned}
$$

[4] By straight-forward we mean that the codomain of the fold is a simple data type, not a function or a tuple.

$$\text{fact}\ n\ =\ \text{Pr}^{int}\ (\lambda().1, \lambda(i, r).i * r)\ n$$

We call such a function for a type constructor T a *primitive recursion* function (paramorphism [13]) for T, since it generalizes the primitive recursion induction scheme of the natural numbers to an arbitrary type.

Definition 9 (Primitive Recursion) *The unary primitive recursive operator over the type T is $\text{Pr}^T(\overline{f})$, and is defined by the following set of recursive equations:*

$$\text{Pr}^T(\overline{f}) \circ C_i = f_i \circ E_i^T(id_1, \ldots, id_p, \langle id, \text{Pr}^T(\overline{f})\rangle)$$

for every constructor C_i of T. Where $\langle f, g\rangle\, x\ =\ (f\ x, g\ x)$.

For example, the list primitive recursive operator is defined as:

$$
\begin{aligned}
\text{Pr}^{list}\ (f_n, f_c)\ \text{Nil} &= f_n()\\
\text{Pr}^{list}\ (f_n, f_c)\ \text{Cons}(a, l) &= f_c(a, (l, \text{Pr}^{list}\ (f_n, f_c)\ l))
\end{aligned}
$$

The promotion theorem for primitive recursion is slightly more complex than the one for fold as there is a relationship between the variables and their recursively transformed copies, both of which appear in the output of $E_i^T(id_1, \ldots, id_p, \langle id, Pr^T(\overline{f})\rangle)$ [7, 6].

Theorem 4 (Primitive Recursion Promotion)

$$\frac{\begin{aligned}\forall i:\ \phi_i \circ E_i^T(id_1, \ldots, id_p, \langle id, g \circ \text{Pr}^T(\overline{f})\rangle)\ &=\\ g \circ f_i \circ E_i^T(id_1, \ldots, id_p, \langle id, \text{Pr}^T(\overline{f})\rangle)&\end{aligned}}{g \circ \text{Pr}^T(\overline{f})\ =\ \text{Pr}^T(\overline{\phi})}$$

Proof: Similar to Theorem 1. □

For example, the promotion theorem for $T\ =\ list$ is:

$$\frac{\begin{aligned}\phi_n() &= g(f_n())\\ \phi_c(a, (l, g(\text{Pr}^{list}(f_n, f_c)\ l))) &= g(f_c(a, (l, \text{Pr}^{list}(f_n, f_c)\ l)))\end{aligned}}{g(\text{Pr}^{list}(f_n, f_c)\ x)\ =\ \text{Pr}^{list}(\phi_n, \phi_c)\ x}$$

The promotion theorem justifies a normalization algorithm for primitive recursions as well. To see this, consider generalizing the term $(\text{Pr}^{list}(f_n, f_c)\ l)$ to the variable r. Thus:

$$\frac{\begin{aligned}\phi_n() &= g(f_n())\\ \phi_c(a, (l, g(r))) &= g(f_c(a, (l, r)))\end{aligned}}{g(\text{Pr}^{list}(f_n, f_c)\ x)\ =\ \text{Pr}^{list}(\phi_n, \phi_c)\ x}$$

This theorem resembles the promotion theorem for folds and can be used to extend the normalization algorithm from folds to primitive recursions in a seamless manner.

7 Second-order Folds

Some common computations are not amenable to normalization since they traverse accumulative result variables. Recall the reverse example from Section 5.1:

$$\text{rev}(x)\ =\ \text{fold}^{list}(\lambda().\text{Nil}, \lambda(a, r).\text{append}(r, \text{Cons}(a, \text{Nil})))\ x$$

where *append* is a fold which traverses the accumulative result variable r. Many such functions can be equivalently expressed by functions which use an extra parameter which acts as an accumulator or a continuation [17]. Such functions can be expressed as second-order folds (so called since the accumulating functions return functions as results). These second-order fold computations are often normalizable although their equivalent first-order computation is not. For example, the iterative version of the list reverse is $\mathrm{rev}(x) = \mathrm{itrev}(x, \mathrm{Nil})$, where *itrev* is computed by:

$$\begin{array}{lcl} \mathrm{itrev}\ \mathrm{Nil}\ w & = & w \\ \mathrm{itrev}\ (\mathrm{Cons}(a,l))\ w & = & \mathrm{itrev}\ l\ (\mathrm{Cons}(a,w)) \end{array}$$

The second-order fold form of this iterative computation is:

$$\mathrm{rev}(x) \;=\; \mathrm{fold}^{list}(\lambda().\lambda w.w, \lambda(a,r).\lambda w.r(\mathrm{Cons}(a,w)))\, x\, \mathrm{Nil}$$

We will see that this form is amenable to normalization.

In this section we describe how such fold computations can be recognized and automatically transformed into their second-order counterparts, which can then be normalized.

Definition 10 ($\mathcal{A}$-function) *A binary function $\mathcal{G}$ is an $\mathcal{A}$-function if it is associative with right identity $\mathcal{I}_\mathcal{G}$, that is*

$$\mathcal{G}(a, \mathcal{G}(b,c)) \;=\; \mathcal{G}(\mathcal{G}(a,b),c) \qquad \text{and} \qquad \mathcal{G}(a, \mathcal{I}_\mathcal{G}) \;=\; a$$

Such functions are not uncommon. In fact every type T which has a zero constructor, C_z (a constructor with no arguments, like Nil for list), has a *zero replacement function* that is associative, and that has the zero, C_z, for both a left and right identity [11]. The zero replacement for T is defined by:

$$\begin{array}{lcl} (\mathrm{Zr}^T\, y) \circ C_z & = & \lambda().y \\ (\mathrm{Zr}^T\, y) \circ C_i & = & C_i \circ E_i^T(id_1, \ldots, id_p, \mathrm{Zr}^T\, y) \end{array}$$

We call $(\mathrm{Zr}^T\, y\, x)$ a zero replacement, since it replaces all zeros in x with y. Note that any zero replacement function $\mathcal{G}(x,y) = (\mathrm{Zr}^T\, y\, x)$ can be expressed as a fold over x. Recognize that Zr^{list} is the list append operator, and that Zr^{int} is integer addition.

Definition 11 ($\mathcal{G}$-term) *Let $\mathcal{G}$ be an $\mathcal{A}$-function and $\overline{r}$ a set of accumulative result variables. A term in our term language is a $\mathcal{G}$-term for the set $\overline{r}$ if it can be expressed using the following rules. 1) An accumulative result variable from $\overline{r}$ is a $\mathcal{G}$-term. 2) A term that does not contain any accumulative result variables from $\overline{r}$ is a $\mathcal{G}$-term. 3) If t and t' are $\mathcal{G}$-terms then the expression $\mathcal{G}(t,t')$ is a $\mathcal{G}$-term.*

Definition 12 ($\mathcal{G}$-fold) *A fold $\mathrm{fold}^T(\overline{f})$ is a $\mathcal{G}$-fold if each accumulating function $f_i(\overline{x_i})$ that introduces the accumulative result variables $\overline{r_i}$ (i.e. $\overline{r_i}$ is a subset of $\overline{x_i}$) can be expressed as a $\mathcal{G}$-term for the set $\overline{r_i}$.*

For example, the reverse function *rev* computed by:

$$\mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{append}(r, \mathrm{Cons}(a, \mathrm{Nil})))\, x$$

is a $\mathcal{G}$-fold for $\mathcal{G} = \mathrm{append}$.

It is a simple matter to syntactically recognize $\mathcal{G}$-fold programs whose accumulating functions are expressed using zero replacement functions. In the following examples we will use only zero replacement functions for $\mathcal{G}$, but the analysis is valid for any associative function $\mathcal{G}$ with a right identity $\mathcal{I}_\mathcal{G}$.

Definition 13 ($\mathcal{F}_\mathcal{G}$-transformation) *For each $\mathcal{A}$-function $\mathcal{G}$ the $\mathcal{F}_\mathcal{G}$-transformation over $\mathcal{G}$-terms is defined as follows:*

$$\begin{array}{lcll} \mathcal{F}_\mathcal{G}(r) & = & \lambda w.r(w) & \text{for accum. result var.} \\ \mathcal{F}_\mathcal{G}(v) & = & \lambda w.\mathcal{G}(v,w) & \text{for value} \\ \mathcal{F}_\mathcal{G}(\mathcal{G}(t,t')) & = & \mathcal{F}_\mathcal{G}(t) \circ \mathcal{F}_\mathcal{G}(t') & \text{otherwise} \end{array}$$

This transformation is similar to CPS transformation that transforms a program in direct style into a program in continuation passing style. Here, instead of a continuation function we use an accumulation value.

Theorem 5 *Let $\mathrm{fold}^T(\overline{f})$ be a $\mathcal{G}$-fold and $\forall i:\ \phi_i = \mathcal{F}_\mathcal{G} \circ f_i$ then*

$$\mathrm{fold}^T(\overline{f})\, x \;=\; \mathrm{fold}^T(\overline{\phi})\, x\, \mathcal{I}_\mathcal{G}$$

Proof: It can be proved by induction that for any x and w:

$$\mathrm{fold}^T(\overline{\phi})\, x\, w \;=\; \mathcal{G}(\mathrm{fold}(\overline{f})\, x,\ w)$$

from which the theorem follows. □

For example, the following non-potentially normalizable first-order fold functions can be transformed to potentially normalizable second-order functions:

$$\begin{array}{l} \mathrm{rev}(x) \\ \quad = \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{append}(r, \mathrm{Cons}(a,\mathrm{Nil})))\, x \\ \quad = \mathrm{fold}^{list}(\lambda().\lambda w.\mathrm{append}(\mathrm{Nil}, w), \\ \qquad\qquad \lambda(a,r).\lambda w.r(\mathrm{append}(\mathrm{Cons}(a,\mathrm{Nil}), w)))\, x\, \mathrm{Nil} \\ \quad = \mathrm{fold}^{list}(\lambda().\lambda w.w, \lambda(a,r).\lambda w.r(\mathrm{Cons}(a,w)))\, x\, \mathrm{Nil} \\ \mathrm{flat}(x) \\ \quad = \mathrm{fold}^{tree}(\lambda(i).\mathrm{Cons}(i,\mathrm{Nil}), \lambda(l,r).\mathrm{append}(l,r))\, x \\ \quad = \mathrm{fold}^{tree}(\lambda(i).\lambda w.\mathrm{append}(\mathrm{Cons}(i,\mathrm{Nil}), w), \\ \qquad\qquad \lambda(l,r).\lambda w.l(r(w)))\, x\, \mathrm{Nil} \\ \quad = \mathrm{fold}^{tree}(\lambda(i).\lambda w.\mathrm{Cons}(i,w), \lambda(l,r).\lambda w.l(r(w)))\, x\, \mathrm{Nil} \\ \mathrm{add\_tips}(x) \\ \quad = \mathrm{fold}^{tree}(\lambda(i).i, \lambda(l,r).\mathrm{plus}(l,r))\, x \\ \quad = \mathrm{fold}^{tree}(\lambda(i).\lambda w.\mathrm{plus}(i,w), \lambda(l,r).\lambda w.l(r(w)))\, x\, \mathrm{Zero} \end{array}$$

In the following theorem we consider folds consisting of sums-of-products types where the recursive type, T, does not appear as a parameter to some other parametric type. That is, we do not consider types such as *bush*. In addition, we simplify the notation of this theorem by assuming that accumulative result variables r_j are separated from the others a_j, such as in $f_i(\overline{a}, \overline{r})$.

Theorem 6 (Second-order Promotion Theorem)

$$\frac{\forall i:\ \bigwedge_j (r_j \circ g \;=\; g \circ s_j) \quad \wedge \quad \phi_i(\overline{a}, \overline{r}) \;=\; f_i(\overline{a}, \overline{s})}{g \circ (\mathrm{fold}^T(\overline{f})\, x) \;=\; (\mathrm{fold}^T(\overline{\phi})\, x) \circ g}$$

We will prove this theorem only for the special case of list second-order folds. For other types the proof is similar.

$$\frac{\begin{array}{c} \phi_n() \circ g \;=\; g \circ f_n() \\ r \circ g \;=\; g \circ s \quad \wedge \quad \phi_c(a,r) \circ g \;=\; g \circ f_c(a,s) \end{array}}{g \circ (\mathrm{fold}^{list}(f_n, f_c)\, x) \;=\; (\mathrm{fold}^{list}(\phi_n, \phi_c)\, x) \circ g}$$

Proof: Base case $x = \mathrm{Nil}$:

$$\begin{array}{lcl} g \circ (\mathrm{fold}^{list}(f_n, f_c)\, \mathrm{Nil}) & = & g \circ f_n() \\ & = & \phi_n() \circ g \\ & = & (\mathrm{fold}^{list}(\phi_n, \phi_c)\, \mathrm{Nil}) \circ g \end{array}$$

Induction hypothesis:

$$g \circ (\mathrm{fold}^{list}(f_n, f_c)\, l) = (\mathrm{fold}^{list}(\phi_n, \phi_c)\, l) \circ g.$$

Induction step $x = \mathrm{Cons}(a,l)$:

$$\begin{array}{lcl} g \circ (\mathrm{fold}^{list}(f_n, f_c)\, \mathrm{Cons}(a,l)) & = & g \circ f_c(a, \mathrm{fold}^{list}(f_n, f_c)\, l) \\ & = & \phi_c(a,r) \circ g \end{array}$$

$$\begin{array}{lcl} \text{where } r \circ g & = & g \circ s \\ & = & g \circ (\mathrm{fold}^{list}(f_n, f_c)\, l) \\ & = & (\mathrm{fold}^{list}(\phi_n, \phi_c)\, l) \circ g \end{array}$$

$\Rightarrow\; r = \mathrm{fold}^{list}(\phi_n, \phi_c)\, l.$

Therefore, $\phi_c(a, r) \circ g \;=\; \phi_c(a, \mathrm{fold}^{list}(\phi_n, \phi_c)\, l) \circ g$
$\;=\; (\mathrm{fold}^{list}(\phi_n, \phi_c)\, \mathrm{Cons}(a, l)) \circ g.\Box$

If $\mathrm{fold}^T(\overline{f})$ is a second-order fold then each accumulating function can be expressed as $f_i(\overline{x}) = \lambda w.e(w)$. We call w an *accumulating stack variable.*

We have developed an extension to the normalization algorithm, based on the second-order promotion theorem, that normalizes second-order folds which contain no folds over accumulative result or accumulating stack variables. This allows the relaxation of the definition of potentially normalizable by allowing folds whose accumulating functions are themselves folds over accumulating result variables, if each such accumulating function is expressed using the same $\mathcal{G}$ function, because we can always apply the $\mathcal{F}_\mathcal{G}$-transform and then apply the extended normalization algorithm.

The following is a sketch of the extended normalization algorithm: When any expression g (that may or may not be a potentially normalizable first-order fold) is applied to a potentially normalizable second-order fold[5], the second-order promotion theorem is used to push g into each accumulating function of that fold. In this process, a subterm $g(s(\ldots))$ is normalized by rewriting it to $r(g(\ldots))$, where s and r are accumulative result variables given in the second-order promotion theorem. The process stops when g is applied to a variable. In that case, g is generalized away and the premise of the promotion theorem is used to derive the new accumulating functions.

For example, $\mathrm{length}(\mathrm{rev}(x))$ is:

$$\mathrm{length}(\mathrm{fold}^{list}(\lambda().\lambda w.w, \lambda(a, r).\lambda w.r(\mathrm{Cons}(a, w)))\, x\, \mathrm{Nil})$$

which according to the promotion theorem can be computed by some:

$$\mathrm{fold}^{list}(\phi_n, \phi_c)\, x\, (\mathrm{length}(\mathrm{Nil})) \;=\; \mathrm{fold}^{list}(\phi_n, \phi_c)\, x\, \mathrm{Zero}$$

We apply the promotion theorem:

$$\begin{aligned}
\phi_n() \circ \mathrm{length} &= \mathrm{length} \circ f_n() \\
&= \mathrm{length} \circ (\lambda w.w) \;=\; \mathrm{length}
\end{aligned}$$
$\Rightarrow \phi_n() \;=\; \lambda w.w$

Assuming $r \circ \mathrm{length} \;=\; \mathrm{length} \circ s$ then

$$\begin{aligned}
\phi_c(a, r) \circ \mathrm{length} &= \mathrm{length} \circ f_c(a, s) \\
&= \mathrm{length} \circ (\lambda w.s(\mathrm{Cons}(a, w))) \\
&= \lambda w.\mathrm{length}(s(\mathrm{Cons}(a, w))) \\
&= \lambda w.r(\mathrm{length}(\mathrm{Cons}(a, w))) \\
&= \lambda w.r(\mathrm{Succ}(\mathrm{length}(w))) \\
&= (\lambda w.r(\mathrm{Succ}(w))) \circ \mathrm{length}
\end{aligned}$$
$\Rightarrow \phi_c(a, r) \;=\; \lambda w.r(\mathrm{Succ}(w))$

Therefore, $\mathrm{length}(\mathrm{rev}(x))$ is:

$$\mathrm{fold}^{list}(\lambda().\lambda w.w, \lambda(a, r).\lambda w.r(\mathrm{Succ}(w)))\, x\, \mathrm{Zero}$$

There are cases where the application of a second-order fold to a second-order fold normalizes to a second-order fold that traverses the accumulating stack variable w. In that case it is necessary to apply the inverse transformation of $\mathcal{F}_\mathcal{G}$ to get a first-order fold from the resulting second-order fold. This is analogous to the DS transformation that translates a program in continuation passing style into a program in direct style.

[5] Note that the application of any function to a potentially improvable first-order fold, such as $\mathrm{reverse}(\mathrm{append}(x, y))$, can always be normalized by the first-order promotion theorem.

Definition 14 (Inverse $\mathcal{F}_\mathcal{G}$-transformation) *The inverse $\mathcal{F}_\mathcal{G}$-transformation $\mathcal{F}_\mathcal{G}^{-1}$ for a second-order fold $\mathrm{fold}^T(\overline{\phi})$ is defined by the following rules:*

$$\begin{aligned}
\mathcal{F}_\mathcal{G}^{-1}(\lambda w.w) &= \mathcal{I}_\mathcal{G} \\
\mathcal{F}_\mathcal{G}^{-1}(\lambda w.r(t)) &= \mathcal{G}(r, \mathcal{F}_\mathcal{G}^{-1}(\lambda w.t)) \\
\mathcal{F}_\mathcal{G}^{-1}(\lambda w.\mathcal{G}(v, t)) &= \mathcal{G}(v, \mathcal{F}_\mathcal{G}^{-1}(\lambda w.t)) \\
\mathcal{F}_\mathcal{G}^{-1}(\lambda w.r(\mathcal{G}(w, t))) &= \mathcal{G}(\mathcal{F}_\mathcal{G}^{-1}(\lambda w.t), r)
\end{aligned}$$

where r is an accumulative result variable of $\mathrm{fold}^T(\overline{\phi})$ (i.e. a variable bound to a function) and v is a value that does not depend on any such accumulative result variable.

The first three rules define the inverse of $\mathcal{G}$-transformation, that is, $\mathcal{F}_\mathcal{G}^{-1} \circ \mathcal{F}_\mathcal{G} = \mathrm{id}$. The last rule is the most important one because it transforms a fold over w (if $\mathcal{G}(w, t)$ is a zero replacement function) into a fold over t. It can be proved that if $f_i = \mathcal{F}_\mathcal{G}^{-1} \circ \phi_i$ then $\mathrm{fold}^T(\overline{\phi})\, x\, \mathcal{I}_\mathcal{G} \;=\; \mathrm{fold}^T(\overline{f})\, x$.

For example,

$$\mathrm{rev}(\mathrm{rev}(x)) \;=\; \mathrm{fold}^{list}(\phi_n, \phi_c)\, x\, \mathrm{rev}(\mathrm{Nil})$$

We apply the promotion theorem:

$\phi_n() \circ \mathrm{rev} \;=\; \mathrm{rev} \circ (\lambda w.w) \;=\; \mathrm{rev}$
$\Rightarrow \phi_n() \;=\; \lambda w.w$

If we assume $r \circ \mathrm{rev} \;=\; \mathrm{rev} \circ s$ then

$$\begin{aligned}
\phi_c(a, r) \circ \mathrm{rev} &= \mathrm{rev} \circ f_c(a, s) \\
&= \mathrm{rev} \circ (\lambda w.s(\mathrm{Cons}(a, w))) \\
&= \lambda w.\mathrm{rev}(s(\mathrm{Cons}(a, w))) \\
&= \lambda w.r(\mathrm{rev}(\mathrm{Cons}(a, w))) \\
&= \lambda w.r(\mathrm{append}(\mathrm{rev}(w), \mathrm{Cons}(a, \mathrm{Nil})))
\end{aligned}$$
$\Rightarrow \phi_c(a, r) \;=\; \lambda w.r(\mathrm{append}(w, \mathrm{Cons}(a, \mathrm{Nil})))$

Therefore, $\mathrm{rev}(\mathrm{rev}(x))$ is

$$\mathrm{fold}^{list}(\lambda().\lambda w.w, \lambda(a, r).\lambda w.r(\mathrm{append}(w, \mathrm{Cons}(a, \mathrm{Nil}))))\, x\, \mathrm{Nil}$$

We apply the $\mathcal{F}_\mathcal{G}^{-1}$ transform to get:

$$\begin{aligned}
&\mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a, r).\mathrm{append}(\mathrm{Cons}(a, \mathrm{Nil}), r))\, x \\
&\quad = \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a, r).\mathrm{Cons}(a, r))\, x
\end{aligned}$$

which is the list identity function.

8 Relationship to Other Work

This work is related to Water's on series expressions [18]. His techniques apply only to linear data structures such as lists, vectors, and streams, yet his techniques automatically detect and fuse multiple walks down the same data structure, unlike some of the more general techniques.

It is also related to Wadler's work on listlessness, and deforestation [14, 15, 8]. Deforestation works on all first order treeless terms. A treeless term is one which is exactly analogous to a *potentially normalizable* term, but is described in a much different manner due to lack of structure imposed on such terms. Wadler makes the observation that some intermediate data structures for primitive types, such as integers, characters, etc. do not really take up space, so he developed a method to handle such terms, using a technique he calls blazing.

Chin's work on fusion [1] extends Wadler's work on deforestation. He generalizes Wadler's techniques to all first order programs, not just treeless ones, by recognizing and skipping over terms to which his techniques do not apply. His work also applies to higher order programs in general. This is accomplished by a higher order removal phase, which first removes some higher order functions from a program. Those not removed are recognizable and are simply "skipped" over in the improvement phase. Our normalization algorithm needs no explicit higher order removal phase.

9 Conclusions

Recursive definitions and higher order functions, as found in most functional programming languages, provide a powerful mechanism for specifying programs, but they do not impose much structure on the form of the programs that may be expressed. By programming with a small, fixed set of recursion patterns derivable from algebraic type definitions, an orderly structure can be imposed upon functional programs. This structure can be exploited to facilitate the proof of program properties, and even to calculate program transformations.

In this paper we generalized the class of types to which these techniques are applicable to mutually recursive sums-of-products with tuples and function types. We described several useful patterns of recursion (*fold*, *derive*, *primitive recursion*, and *second-order fold* in varying levels of detail) applicable to every type in this class.

In addition, we described the *normalization algorithm* which automatically calculates normalized forms for programs expressed in the *fold* sub-language. This algorithm, based upon a generic *promotion theorem*, is facilitated by the explicit structure of fold programs rather than using an analysis phase to search for implicit structure.

We identified generic *promotion theorems* for each pattern of recursion. It is our *hope* that these theorems can be the basis of normalization algorithms for patterns of recursion other than $fold$. At the moment this is an open problem.

Other problems remain as well. Many algorithms can be expressed using several different patterns of recursion. In a system with normalization algorithms for several patterns of recursion, it would be useful to know sufficient conditions for translating such algorithms from one pattern to another. Thus enabling the improvement of terms, not normalizable in one pattern, but normalizable in another.

References

[1] W. Chin. Safe Fusion of Functional Expressions. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, Ca., June 1992.

[2] J. Cockett and D. Spencer. Strong Categorical Datatypes I. In R. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings, Vol. 13, pp 141-169. AMS, Montreal, 1992.

[3] J. Cockett and T. Fukushima. About Charity The University of Calgary, Department of Computer Science, Research Report No. 92/480/18. June 1992.

[4] J. Darlington and R. Burstall. A System which Automatically Improves Programs. *Acta Informatica*, 6(1):41–60, 1976.

[5] H. Dybkjær. Category Theory, Types, and Programming Languages. Ph.D. thesis, Department of Computer Science, University of Copenhagen (DIKU), May 1991.

[6] L. Fegaras. *A Transformational Approach to Database System Implementation.* Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst, February 1993. Also appeared as CMPSCI Technical Report 92-68.

[7] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In *Proceedings of the 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, New York*, pp 148–162. Springer-Verlag, June 1992.

[8] A. Ferguson, and P. Wadler. When will Deforestation Stop. In *Proc. of 1988 Glasgow Workshop on Functional Programming* (also as research report 89/R4 of Glasgow University), pp 39-56, Rothesay, Isle of Bute, August 1988.

[9] T. Hagino. *A Categorical Programming Language.* Ph.D. thesis, University of Edinburgh, 1987.

[10] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[11] J. Hook, R. Kieburtz, and T. Sheard. Generating Programs by Reflection. Oregon Graduate Institute Technical Report 92-015, submitted to *Journal of Functional Programming.*

[12] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pp 335–347. Springer-Verlag, June 1989.

[13] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144, August 1991.

[14] P. Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-time. In *Proc. of the ACM Symposium on Lisp and Functional Programming*, Austin Texas, August, 1984.

[15] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, March 1988. Lecture Notes in Computer Science 300.

[16] P. Wadler. Editorial - Lazy Functional Programming *The Computer Journal*, Vol. 32, No. 2, 1989, p. 97.

[17] M. Wand. Continuation-Based Program Transformation Strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[18] R. Waters. Automatic Transformation of Series Expressions into Loops, *ACM Transactions on Programming Languages and Systems*, 13(1):52-98, January 1991.