Improving Programs which Recurse over Multiple Inductive Structures

Leonidas Fegaras Tim Sheard Tong Zhou

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
{fegaras,sheard,tzhou}@cse.ogi.edu

Abstract

This paper considers generic recursion schemes for programs which recurse over multiple inductive structures simultaneously, such as equality, zip and the nth element of a list function. Such schemes have been notably absent from previous work. This paper defines a uniform mechanism for defining such programs and shows that these programs satisfy generic theorems. These theorems are the basis for an automatic improvement algorithm. This algorithm is an improvement over the algorithm presented earlier [14] because, in addition to inducting over multiple structures, it can be incorporated into any algebraic language and is no longer restricted to a "safe" subset.

1 Introduction

In previous work [14, 15, 6, 4, 5] we have shown how programming algebraically with *generic recursion schemes* provides a theory amenable to program calculation [13]. This theory provides a basis for automatic optimization techniques which capture many well-known transformations. Unfortunately, these recursion schemes may induct over only one structure at a time, and thus cannot capture in a straightforward manner such common functions as *structural equality*, *zip*, or the function that computes the *nth* element of a list.

In this paper we first generalize the generic reduction scheme to induct over any number of structures (not necessarily of the same type) simultaneously. We show that this induction scheme has a generic promotion theorem, and that the theorem supports a normalization algorithm that automatically calculates a number of previously unrelated improvements to programs, such as deforestation, loop fusion, and partial evaluation. This is an important step towards an automatic optimization phase in compilers of algebraic programs.

Second we extend our earlier work [14, 6] by embedding our optimizations in a richer language. Our previous work focused on a restricted language which has now been extended to include many features of a modern functional programming language.

Third we describe an improved algorithm for computing our optimizations. The improved algorithm works over the entire extended language while the original algorithm worked only on a syntactically identifiable subset of the restricted language. In addition the improved algorithm requires no explicit memoization phase (what we called generalization in [14, 6].)

To appear in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, June 1994.

1.1 Goals

Recursion is the *Goto* of functional programming. Languages which allow arbitrary recursive programs lack the structure necessary for automatic optimization. Much recent research has focused on the design of programming systems whose control structures are exclusively some generic recursion schemes [3, 9, 10]. We call this programming style *algebraic programming* because of its reliance on algebras and combinators for encoding control. Notable absent from these works are generic recursion schemes for inducting over multiple structures.

A goal of this paper is to describe internal program representations for algebraic programs which are amenable to completely automatic optimization and transformation methods, yet are expressive enough to encode algorithms which induct over multiple structures simultaneously. These program representations are suitable for use in the back-end of a compiler for an algebraic programming language.

A second goal is to describe how ordinary recursive programs could be translated into this internal form, so that our techniques could be applied to traditional functional languages. In this work we do not consider user interfaces to algebraic programming languages, but rather demonstrate that algebraic recursion schemes are a practical internal form for representing programs and describe a single automatic mechanism for performing a wide range of optimizations over such programs.

1.2 Motivation and Background

Consider for example the following inductive type equation that captures natural numbers:

$$\mathrm{nat} \;=\; \mathrm{Zero} \mid \mathrm{Succ}\ \mathbf{of}\ \mathrm{nat}$$

The *reduction* over a *nat* k can be computed by $\text{red}^{nat}(f_z, f_s)\, k$, where f_z and f_s are functions associated with the value constructors Zero and Succ. The combinator $\text{red}^{nat}(f_z, f_s)$ is defined as follows:

$$\begin{array}{lcl} \text{red}^{nat}(f_z, f_s)\,\text{Zero} & = & f_z() \\ \text{red}^{nat}(f_z, f_s)\,(\text{Succ}(n)) & = & f_s(\text{red}^{nat}(f_z, f_s)\, n) \end{array}$$

Functions f_z and f_s are called *accumulating functions.*

For example:

$$x + y \;=\; \text{red}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\, x$$

The variable r in $\lambda(r)$ is the partial result of the reduction, since it represents the value of the recursive call $\text{red}^{nat}(f_z, f_s)\, n$. It is called an *accumulative result variable.*

The reduction operator can be generalized for most inductively defined data type. One such type is *list*:

$$\text{list}(\alpha) \;=\; \text{Nil} \mid \text{Cons}\ \mathbf{of}\ \alpha \times \text{list}(\alpha)$$

where α is a type variable. The reduction for lists is very similar to the reduction for natural numbers:

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

For example,

$$\text{length}(x) \;=\; \text{red}^{list}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r))\, x$$

For each such inductive data type there is a generic theorem called the *promotion theorem* [12, 13]. For lists this theorem takes the following form:

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

This states that the application of any unary function g to a reduction over a list x is another reduction over the same list x whose accumulating functions ϕ_n and ϕ_c are related to the original accumulating functions f_n and f_c by the two equations in the premise of the theorem. The first equation calculates directly a solution for ϕ_n in terms of g and f_n. The second equation, though, cannot be solved directly. This equation must be rearranged so that the term $g(r)$ can be generalized to a variable in both sides of the equation. That is, the term $g(f_c(a, r))$ must be transformed into a form $\mathcal{T}(g(r))$, for some term $\mathcal{T}$ that depends on $g(r)$ but not on r. Such work was previously reported in [14, 6, 5, 4] where this transformation is achieved by the generalization phase of the *normalization algorithm.*

As an example, we will improve $\text{length}(\text{append}(x, y))$, where

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

We need to find some $\text{red}^{list}(\phi_n, \phi_c)\, x = \text{length}(\text{append}(x, y))$. We apply the *list* promotion theorem with $g = \text{length}$ and $\text{red}^{list}(f_n, f_c)\, x = \text{append}(x, y)$:

$$\begin{array}{lllcl} 1) & \phi_n() & = & g(f_n()) \;=\; \text{length}(f_n()) \\ & & = & \text{length}(y) \\ 2) & \phi_c(a, \text{length}(r)) & = & g(f_c(a, r)) \;=\; \text{length}(f_c(a, r)) \\ & & = & \text{length}(\text{Cons}(a, r)) \\ & & = & \text{Succ}(\text{length}(r)) \\ & & & \textit{(by the length definition)} \\ & \Rightarrow\ \phi_c(a, u) \;=\; \text{Succ}(u) & & \\ & \textit{(where } \text{length}(r) \textit{ was generalized to } u\textit{)} & & \end{array}$$

Therefore, $\text{length}(\text{append}(x, y))$ is transformed into:

$$\text{red}^{list}(\lambda().\text{length}(y), \lambda(a, u).\text{Succ}(u))\, x$$

Note that $\text{length}(\text{append}(x, y))$ generates an intermediate data structure (a list), since $\text{append}(x, y)$ constructs a list which is consumed by length. This data structure is not produced when this composition is normalized into the reduction above.

Even though the reduction scheme that inducts over one value (henceforth called a unary reduction) is very powerful, there are still some important functions that cannot be captured directly by this mechanism. One class are binary functions such as structural equality. For example, the structural equality over lists is defined as follows:

$$\begin{array}{lcl} \text{listeq}(\text{Nil}, \text{Nil}) & = & \text{True} \\ \text{listeq}(\text{Nil}, \text{Cons}(b, s)) & = & \text{False} \\ \text{listeq}(\text{Cons}(a, l), \text{Nil}) & = & \text{False} \\ \text{listeq}(\text{Cons}(a, l), \text{Cons}(b, s)) & = & (a = b) \wedge \text{listeq}(l, s) \end{array}$$

Function listeq cannot be expressed directly as a unary reduction since it needs to walk through the two input lists simultaneously. We can express $\text{listeq}(x, y)$ as a second-order unary reduction though:

$$\begin{array}{l} \text{red}^{list}(\lambda().\lambda k.(k = \text{Nil}), \\ \qquad \lambda(a, r).\lambda k.\mathbf{case}\ k\ \mathbf{of}\ \text{Nil} \Rightarrow \text{False} \\ \qquad\qquad\qquad\qquad \mid \text{Cons}(b, s) \Rightarrow (a = b) \wedge r(s))\, x\, y \end{array}$$

but this form does not facilitate automated optimization. In particular, this reduction inducts over the first input x but not over y. The accumulative result variable r is now a function that "recurses" over the second parameter y. That is, this program is both inductive and higher-order and relies on the case expression to decompose y one level at a time, while the "recursion" inherent in the accumulative result variable r drives the induction over y. The optimization techniques described in the next section are applicable only to the direct object of the fold, x, but not to y. We can write another similar program where the direct object of the fold is y and the higher-order value returned by this fold analyses x. The lack of symmetry in these folds motivates simultaneous multiple induction schemes.

In this paper we capture functions, such as listeq, using a new reduction scheme $F = \text{red}^{list \times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})$, defined as follows[1]:

$$\begin{array}{lcl} F(\text{Nil}, \text{Nil}) & = & f_{nn}((), ()) \\ F(\text{Nil}, \text{Cons}(b, s)) & = & f_{nc}((), (b, s)) \\ F(\text{Cons}(a, l), \text{Nil}) & = & f_{cn}((a, l), ()) \\ F(\text{Cons}(a, l), \text{Cons}(b, s)) & = & f_{cc}(a, (b, F(l, s))) \end{array}$$

[1] The unintuitive "shape" of the domain of the accumulating functions will be explained later.

In that case, listeq(x, y) is equal to:

$$\begin{aligned}\text{red}^{list\times list}(&\lambda((),()).\text{True},\\ &\lambda((),(b,s)).\text{False},\\ &\lambda((a,l),()).\text{False},\\ &\lambda(a,(b,r)).(a=b)\wedge r)\,(x,y)\end{aligned}$$

We call $\text{red}^{list\times list}$ a binary reduction because it inducts over two data structures simultaneously.

This paper extends the theory of unary reductions to capture all reduction schemes that induct over any number of data structures, which need not necessarily be of the same type. We will present and prove a very general promotion theorem that captures all types of compositions between these general reduction schemes. We will use this theorem as the basis of a very effective and efficient normalization algorithm that improves most programs in our term language. This algorithm eliminates intermediate data structures, which might be generated when reductions are nested, passing intermediate results from one to another.

Finally, we demonstrate that the same algorithm that eliminates intermediate data structures can be adapted to translate ordinary recursive function definitions into their reduction-based counterparts.

This paper is organized as follows. Section 2 reviews the definition of the unary reduction operator for any inductively defined data type. These combinators are then generalized in such a way that they can be used for defining reductions that induct over multiple data types. Section 3 presents two instances of the promotion theorem, one is for promoting a unary function and the other for promoting a binary function. The promotion theorem in its most general form is presented in Section 5. Section 4 presents the normalization algorithm for the unary and binary case. Finally, Section 6 presents an automated method for converting regular ML-style recursive function definitions into algebraic programs.

2 Reductions

The type definitions considered in this paper are the inductive types defined by using recursive equations of the form:

$$T(\alpha_1,\dots,\alpha_p) \;=\; C_1 \textbf{ of } t_1 \;\mid\; \cdots \;\mid\; C_n \textbf{ of } t_n$$

where $\alpha_1,\dots,\alpha_p$ denote type variables (abbreviated by the vector $\overline{\alpha}$), the C_i are names of value constructor functions, and each type t_i is an inductive subcomponent of type $T(\overline{\alpha})$. An *inductive subcomponent* of a type $T(\overline{\alpha})$ has one of the following forms:

α_i	a type variable in the set $\alpha_1,\dots,\alpha_p$
$()$	the unit type
$t_1\times t_2$	a pair of two inductive subcomponents of $T(\overline{\alpha})$
$T(\overline{\alpha})$	the recursive reference to $T(\overline{\alpha})$
$S(t_1,\dots,t_q)$	where S is a previously defined inductive type constructor and each t_i is an inductive subcomponent of $T(\overline{\alpha})$

For example, the following are inductive type definitions:

$$\begin{aligned}\text{boolean} &= \text{False} \mid \text{True}\\ \text{list}(\alpha) &= \text{Nil} \mid \text{Cons} \textbf{ of } \alpha\times\text{list}(\alpha)\\ \text{tree}(\alpha,\beta) &= \text{Tip} \textbf{ of } \alpha \mid \text{Node} \textbf{ of } \beta\times\text{tree}(\alpha,\beta)\times\text{tree}(\alpha,\beta)\\ \text{bush}(\alpha) &= \text{Leaf} \textbf{ of } \alpha \mid \text{Branch} \textbf{ of } \text{list}(\text{bush}(\alpha))\end{aligned}$$

Definition 1 (The Functor E) *For each value constructor C of type $t\rightarrow T(\overline{\alpha})$ we associate a functor E_c^T, such that $E_c^T(f) = \mathcal{K}[T(\overline{\alpha}),t](f,\text{id},\dots,\text{id})$, where* id *is the identity function. The combinator $\mathcal{K}[T(\alpha_1,\dots,\alpha_p),t](f,g_1,\dots,g_p)$ is defined by the following inductive equations:*

$$\begin{aligned}\mathcal{K}[T(\overline{\alpha}),\alpha_i](f,\overline{g}) &= g_i\\ \mathcal{K}[T(\overline{\alpha}),()](f,\overline{g}) &= \text{id}\\ \mathcal{K}[T(\overline{\alpha}),t_1\times t_2](f,\overline{g}) &= \mathcal{K}[T(\overline{\alpha}),t_1](f,\overline{g})\times\mathcal{K}[T(\overline{\alpha}),t_2](f,\overline{g})\\ \mathcal{K}[T(\overline{\alpha}),T(\overline{\alpha})](f,\overline{g}) &= f\\ \mathcal{K}[T(\overline{\alpha}),S(t_1,\dots,t_q)](f,\overline{g})&\\ &= \text{map}^S(\mathcal{K}[T(\overline{\alpha}),t_1](f,\overline{g}),\dots,\mathcal{K}[T(\overline{\alpha}),t_q](f,\overline{g}))\end{aligned}$$

where the product of two functions g and h is defined by $(g\times h)(x,y) = (g\,x, h\,y)$ and map^S is a map over the inductive type $S(\alpha_1,\dots,\alpha_q)$, i.e., it maps the parametric type $S(\alpha_1,\dots,\alpha_q)$ into the type $S(\beta_1,\dots,\beta_q)$. That is, for each value constructor C of type $t\rightarrow T(\overline{\alpha})$ we have:

$$\text{map}^T(\overline{f})\circ C \;=\; C\circ\mathcal{K}[T(\overline{\alpha}),t](\text{map}^T(\overline{f}),\overline{f})$$

where $\circ$ is function composition (i.e. $(f\circ g)\,x = f(g(x))$). For example, the map for *list* is:

$$\begin{aligned}\text{map}^{list}(f)\,\text{Nil} &= \text{Nil}\\ \text{map}^{list}(f)\,(\text{Cons}(a,l)) &= \text{Cons}(f\,a,\text{map}^{list}(f)\,l)\end{aligned}$$

It is easy to prove that $E_c^T(\text{id}) = \text{id}$ and $E_c^T(f\circ g) = E_c^T(f)\circ E_c^T(g)$. That is, E_c^T is a functor.

For example:

$$\begin{aligned}E_{Nil}(f) &= \text{id} &&= \lambda().()\\ E_{Cons}(f) &= \text{id}\times f &&= \lambda(a,s).(a,f(s))\\ E_{Node}(f) &= \text{id}\times f\times f &&= \lambda(i,l,r).(i,f(l),f(r))\\ E_{Branch}(f) &= \text{map}^{list}(f) &&= \lambda(l).\text{map}^{list}(f)\,l\end{aligned}$$

Definition 2 (Unary Reduction) *The unary reduction operator over the type T is $\text{red}^T(\overline{f})$ and it is defined by the following set of recursive equations, one for each value constructor C of T:*

$$\text{red}^T(\overline{f})\circ C = f_c\circ E_c^T(\text{red}^T(\overline{f}))$$

where variable $\overline{f}$ is the vector of all accumulating functions f_c for each value constructor C of T.

For example, the *tree* reduction operator is defined as:

$$\begin{aligned}\text{red}^{tree}(f_t,f_n)\,(\text{Tip}(a)) &= f_t(a)\\ \text{red}^{tree}(f_t,f_n)\,(\text{Node}(b,l,r))&\\ &= f_n(b,\text{red}^{tree}(f_t,f_n)\,l,\text{red}^{tree}(f_t,f_n)\,r)\end{aligned}$$

The *bush* reduction operator is defined as:

$$\begin{aligned}\text{red}^{bush}(f_l,f_b)\,(\text{Leaf}(a)) &= f_l(a)\\ \text{red}^{bush}(f_l,f_b)\,(\text{Branch}(l)) &= f_b(\text{map}^{list}(\text{red}^{bush}(f_l,f_b))\,l)\end{aligned}$$

The following are some examples of computations that use unary reduction operators:

$$
\begin{array}{lcl}
\text{append}(x,y) & = & \text{red}^{list}(\lambda().y, \lambda(a,r).\text{Cons}(a,r))\, x \\
\text{length}(x) & = & \text{red}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\, x \\
\text{reverse}(x) & = & \text{red}^{list}(\lambda().\text{Nil}, \\
 & & \quad \lambda(a,r).\text{append}(r, \text{Cons}(a,\text{Nil})))\, x \\
x+y & = & \text{red}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\, x \\
\text{if}\, x\, \text{then}\, y\, \text{else}\, z & = & \text{red}^{boolean}(\lambda().z, \lambda().y)\, x \\
\text{sum\_tree}(x) & = & \text{red}^{tree}(\lambda(a).a, \lambda(b,m,n).b+m+n)\, x \\
\text{reflect\_bush}(x) & = & \text{red}^{bush}(\lambda(a).\text{Leaf}(a), \\
 & & \quad \lambda(r).\text{Branch}(\text{reverse}(r)))\, x
\end{array}
$$

In order to generalize reduction to capture recursion schemas that traverse more than one data structure simultaneously, we need to generalize the functor E.

Definition 3 (Generalized Product Type) *A generalized product type τ is either an inductive type T or a pair $\tau_1 \times \tau_2$, where τ_1 and τ_2 are generalized product types.*

We use the symbol T for inductive types and τ for generalized product types.

Definition 4 (Generalized Constructor) *The set of generalized constructors $\mathcal{GC}(\tau)$ of a generalized product type τ is defined inductively using list comprehensions:*

$$
\begin{array}{rcl}
\mathcal{GC}(T) & = & [C \mid C \text{ is a value constructor of } T] \\
\mathcal{GC}(\tau_1 \times \tau_2) & = & [c_1 \times c_2 \mid c_1 \leftarrow \mathcal{GC}(\tau_1),\, c_2 \leftarrow \mathcal{GC}(\tau_2)]
\end{array}
$$

We will use the symbols C, C_1, or C_2 for value constructors and the symbols c, c_1, or c_2 for generalized constructors.

For example, the generalized constructors for $\tau(\alpha) = \text{list}(\alpha) \times \text{nat}$ are: Nil $\times$ Zero, Nil $\times$ Succ, Cons $\times$ Zero, and Cons$\times$Succ. The generalized constructors for $\tau(\alpha) = \text{boolean} \times (\text{list}(\alpha) \times \text{nat})$ are False $\times$ (Nil $\times$ Zero), False $\times$ (Nil $\times$ Succ), False $\times$ (Cons $\times$ Zero), False $\times$ (Cons $\times$ Succ), True $\times$ (Nil $\times$ Zero), True $\times$ (Nil $\times$ Succ), True $\times$ (Cons $\times$ Zero), and True $\times$ (Cons $\times$ Succ).

Definition 5 (Inductive Constructor) *A generalized constructor $c \in \mathcal{GC}(\tau)$ is* inductive *(denoted as* inductive(c)*) if either c is the value constructor C of type $t \rightarrow T$ and type t (the domain of C) contains a reference to T, or $c = c_1 \times c_2$ and both c_1 and c_2 are inductive constructors.*

That is, a generalized constructor c of τ is not inductive if the domain of any of its constituent value constructors $C : t \rightarrow T$ has no recursive reference to type T. For example, Cons $\times$ Succ is inductive while Cons $\times$ Zero is not.

The following combinators are defined in terms of the E functor and they are, in a way, generalizations of E:

Definition 6 (The Functor $\mathcal{D}$)

$$
\begin{array}{lcll}
\mathcal{D}_c^T(f) & = & \text{id} & \text{if } \neg\text{inductive}(c) \\
\mathcal{D}_c^T(f) & = & E_c^T(f) & \rbrace \text{ if inductive}(c) \\
\mathcal{D}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(f) & = & \mathcal{D}_{c_1}^{\tau_1}(\mathcal{D}_{c_2}^{\tau_2}(f)) &
\end{array}
$$

Note that $\mathcal{D}$ is a functor since it is a composition of functors.

Definition 7 (The Combinator $\mathcal{E}$)

$$
\begin{array}{lcll}
\mathcal{E}_c^T(f) & = & \text{id} & \text{if } \neg\text{inductive}(c) \\
\mathcal{E}_c^T(f) & = & E_c^T(f) & \text{if inductive}(c) \\
\mathcal{E}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(f) & = & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.f(z,w))\, y)\, x &
\end{array}
$$

The general intuition behind this definition can be explained as follows. Let C_1, C_2, and f have the typings:

$$
\begin{array}{rl}
C_1 : & (\alpha \times T_1(\alpha) \times \alpha) \rightarrow T_1(\alpha) \\
C_2 : & (T_2(\beta) \times \beta \times T_2(\beta)) \rightarrow T_2(\beta) \\
f : & (T_1(\alpha) \times T_2(\beta)) \rightarrow \gamma
\end{array}
$$

then $\mathcal{E}_{C_1 \times C_2}(f)$ has type:

$$(\alpha \times T_1(\alpha) \times \alpha) \times (T_2(\beta) \times \beta \times T_2(\beta)) \rightarrow (\alpha \times (\gamma \times \beta \times \gamma) \times \alpha)$$

That is, if $\mathcal{E}_{C_1 \times C_2}(f)$ is applied to $((a,x,b),(y_1,c,y_2))$, then it returns $(a,(g(x,y_1),c,g(x,y_2)),b)$. This returned tuple has the same shape as the domain of the first constructor C_1, but any components of type T_1 in this tuple are replaced by tuples that have the same shape as the domain of the second constructor C_2. That is, x in (a,x,b) is replaced by (y_1,c,y_2) (since the type of x is $T_1(\alpha)$) and each y_i in $(a,(y_1,c,y_2),b)$ is replaced by $g(x,y_i)$.

For example:

$$
\begin{array}{lcl}
\mathcal{D}_{Nil \times Tip}(f) & = & \text{id} \;=\; \lambda((),i).((),i) \\
\mathcal{E}_{Nil \times Tip}(f) & = & \text{id} \;=\; \lambda((),i).((),i) \\
\mathcal{D}_{Cons \times Tip}(f) & = & \text{id} \;=\; \lambda((a,s),i).((a,s),i) \\
\mathcal{E}_{Cons \times Tip}(f) & = & \text{id} \;=\; \lambda((a,s),i).((a,s),i) \\
\mathcal{D}_{Cons \times Cons}(f) & = & \lambda(x,(y,r)).(x,(y,f(r))) \\
\mathcal{E}_{Cons \times Cons}(f) & = & \lambda((x,xs),(y,ys)).(x,(y,f(xs,ys))) \\
\mathcal{D}_{Cons \times Succ}(f) & = & \lambda(x,r).(x,f(r)) \\
\mathcal{E}_{Cons \times Succ}(f) & = & \lambda((x,xs),n).(x,f(xs,n)) \\
\mathcal{D}_{Cons \times Node}(f) & = & \lambda(a,(i,r_1,r_2)).(a,(i,f(r_1),f(r_2))) \\
\mathcal{E}_{Cons \times Node}(f) & = & \lambda((a,s),(i,l,r)).(a,(i,f(s,l),f(s,r))) \\
\mathcal{D}_{Node \times Node}(f) & = & \lambda(i,(j,r_1,r_2),(k,r_3,r_4)).(i,(j,f(r_1), \\
 & & \quad f(r_2)),(k,f(r_3),f(r_4))) \\
\mathcal{E}_{Node \times Node}(f) & = & \lambda((i,l,r),(j,m,n)).(i,(j,f(l,m), \\
 & & \quad f(l,n)),(j,f(r,m),f(r,n)))
\end{array}
$$

Definition 8 (The Combinator $\mathcal{M}$)

$$
\begin{array}{lcl}
\mathcal{M}_c^T(f) & = & \text{id} \qquad\qquad \text{if } \neg\text{inductive}(c) \\
\mathcal{M}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(f) & = & \lambda(x,y).\mathcal{D}_{c_1}^{\tau_1}(\lambda z.\mathcal{D}_{c_2}^{\tau_2}(\lambda w.f(z,w))\, y)\, x \\
 & & \qquad\qquad \text{if inductive}(c)
\end{array}
$$

Note that if τ_1 and τ_2 are the simple inductive types T_1 and T_2 respectively, then $\mathcal{M}_{c_1 \times c_2}^{T_1 \times T_2}(f) = \mathcal{E}_{c_1 \times c_2}^{T_1 \times T_2}(f)$.

Properties:

$$
\begin{array}{rcll}
\mathcal{D}_c^T(\text{id}) & = & \text{id} & (1) \\
\mathcal{D}_c^T(f \circ g) & = & \mathcal{D}_c^T(f) \circ \mathcal{D}_c^T(g) & (2) \\
\mathcal{E}_c^T(f \circ g) & = & \mathcal{D}_c^T(f) \circ \mathcal{E}_c^T(g) & (3) \\
\mathcal{E}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(g \circ (f \times h)) & & & \\
 & = & \mathcal{M}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(g) \circ (\mathcal{E}_{c_1}^{\tau_1}(f) \times \mathcal{E}_{c_2}^{\tau_2}(h)) & (4)
\end{array}
$$

Proof: Properties 1 and 2 are true because $\mathcal{D}$ is a functor (since it is a composition of functors). Property 3 is true for a non-inductive c and for $\tau = T$. We assume that it is true for $\tau = \tau_1$ and $\tau = \tau_2$ (induction hypothesis). Then for $\tau = \tau_1 \times \tau_2$ and $c = c_1 \times c_2$ we have:

$$
\begin{aligned}
& \mathcal{D}_c^\tau(f) \circ \mathcal{E}_c^\tau(g) \\
=\ & \mathcal{D}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(f) \circ \mathcal{E}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(g) \\
=\ & \mathcal{D}_{c_1}^{\tau_1}(\mathcal{D}_{c_2}^{\tau_2}(f)) \circ (\lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,y)\,x) \\
& \qquad \text{(by Definitions 6 and 7)} \\
=\ & \lambda(x,y).\mathcal{D}_{c_1}^{\tau_1}(\mathcal{D}_{c_2}^{\tau_2}(f))(\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,y)\,x) \\
=\ & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}((\mathcal{D}_{c_2}^{\tau_2}(f) \circ (\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,y))\,x) \\
& \qquad \text{(induction hypothesis)} \\
=\ & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{D}_{c_2}^{\tau_2}(f)(\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,y)\,x) \\
=\ & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}((f \circ (\lambda w.g(z,w)))\,y)\,x) \\
& \qquad \text{(induction hypothesis)} \\
=\ & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.f(g(z,w)))\,y)\,x \\
=\ & \mathcal{E}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(f \circ g) \qquad \text{(by Definition 7)} \\
=\ & \mathcal{E}_c^\tau(f \circ g)
\end{aligned}
$$

Property 4 can be proved as follows:

$$
\begin{aligned}
& \mathcal{M}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(g) \circ (\mathcal{E}_{c_1}^{\tau_1}(f) \times \mathcal{E}_{c_2}^{\tau_2}(h)) \\
=\ & (\lambda(x,y).\mathcal{D}_{c_1}^{\tau_1}(\lambda z.\mathcal{D}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,y)\,x) \circ (\mathcal{E}_{c_1}^{\tau_1}(f) \times \mathcal{E}_{c_2}^{\tau_2}(h)) \\
& \qquad \text{(by Definition 8)} \\
=\ & \lambda(x,y).\mathcal{D}_{c_1}^{\tau_1}(\lambda z.\mathcal{D}_{c_2}^{\tau_2}(\lambda w.g(z,w))\,(\mathcal{E}_{c_2}^{\tau_2}(h)\,y))\,(\mathcal{E}_{c_1}^{\tau_1}(f)\,x) \\
& \qquad \text{(by beta reduction)} \\
=\ & \lambda(x,y).\mathcal{D}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(z,h\,w))\,y)\,(\mathcal{E}_{c_1}^{\tau_1}(f)\,x) \\
& \qquad \text{(by Property 3)} \\
=\ & \lambda(x,y).\mathcal{E}_{c_1}^{\tau_1}(\lambda z.\mathcal{E}_{c_2}^{\tau_2}(\lambda w.g(f\,z,h\,w))\,y)\,x \\
& \qquad \text{(by Property 3)} \\
=\ & \mathcal{E}_{c_1 \times c_2}^{\tau_1 \times \tau_2}(g \circ (f \times h)) \qquad \text{(by Definition 7)} \qquad \Box
\end{aligned}
$$

We are now ready to define the generalized reduction scheme:

Definition 9 (Reduction)

$$\forall c \in \mathcal{GC}(\tau) : \ \text{red}^\tau(\overline{f}) \circ c \ = \ f_c \circ \mathcal{E}_c^\tau(\text{red}^\tau(\overline{f}))$$

For example, the following is the definition of the binary reduction:

$$\text{red}^{T_1 \times T_2}(\overline{f}) \circ (C_1 \times C_2) \ = \ f_{c_1 \times c_2} \circ \mathcal{E}_{c_1 \times c_2}^{T_1 \times T_2}(\text{red}^{T_1 \times T_2}(\overline{f}))$$

where C_1 and C_2 are value constructors of T_1 and T_2, respectively.

For example, the binary reduction operator

$$F \ = \ \text{red}^{list \times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})$$

is defined as:

$$
\begin{aligned}
F(\text{Nil}, \text{Nil}) &= f_{nn}((), ()) \\
F(\text{Nil}, \text{Cons}(b,s)) &= f_{nc}((), (b,s)) \\
F(\text{Cons}(a,l), \text{Nil}) &= f_{cn}((a,l), ()) \\
F(\text{Cons}(a,l), \text{Cons}(b,s)) &= f_{cc}(a, (b, F(l,s)))
\end{aligned}
$$

The following are examples of binary reductions:

$$
\begin{aligned}
\text{nateq}(x,y) &= \text{red}^{nat \times nat}(\lambda((),()).\text{True},\ \lambda((),j).\text{False}, \\
& \qquad \lambda(i,()).\text{False},\ \lambda(r).r)\,(x,y) \\
\text{listeq}(x,y) &= \text{red}^{list \times list}(\lambda((),()).\text{True},\ \lambda((),(b,s)).\text{False}, \\
& \qquad \lambda((a,l),()).\text{False}, \\
& \qquad \lambda(a,(b,r)).r \wedge (a = b))\,(x,y) \\
\text{zip}(x,y) &= \text{red}^{list \times list}(\lambda((),()).\text{Nil},\ \lambda((),(b,s)).\text{Nil}, \\
& \qquad \lambda((a,l),()).\text{Nil}, \\
& \qquad \lambda(a,(b,r)).\text{Cons}((a,b),r))\,(x,y) \\
\text{firstn}(n,x) &= \text{red}^{nat \times list}(\lambda((),()).\text{Nil},\ \lambda((),(a,l)).\text{Nil}, \\
& \qquad \lambda(i,()).\text{Nil}, \\
& \qquad \lambda(a,r).\text{Cons}(a,r))\,(n,x) \\
\text{nth}(d)(x,n) &= \text{red}^{list \times nat}(\lambda((),()).d,\ \lambda((),i).d, \\
& \qquad \lambda((a,l),()).a,\ \lambda(a,r).r)\,(x,n) \\
x \dot{-} y &= \text{red}^{nat \times nat}(\lambda((),()).\text{Zero},\ \lambda((),j).\text{Zero}, \\
& \qquad \lambda(i,()).\text{Succ}(i),\ \lambda(r).r)\,(x,y)
\end{aligned}
$$

3 Promotion Theorems

The general law which applies to all reductions is called the *general promotion theorem*. In this section we will present two special cases of the general promotion theorem, which are also the most common cases. The promotion theorem is presented and proved in its general form in Section 5. These are the unary and the binary promotion theorems (a unary function composed with one generalized reduction and a binary function composed with two generalized reductions).

The first promotion theorem is for the case of composing a unary function g with any n-ary reduction.

Theorem 1 (Unary Promotion Theorem)

$$\frac{\forall c \in \mathcal{GC}(\tau) : \ \phi_c \circ \mathcal{D}_c^\tau(g) \ = \ g \circ f_c}{g \circ \text{red}^\tau(\overline{f}) \ = \ \text{red}^\tau(\overline{\phi})}$$

Proof: Let $\eta = g \circ \text{red}^\tau(\overline{f})$ and $c \in \mathcal{GC}(\tau)$. Then

$$
\begin{aligned}
\eta \circ c &= g \circ \text{red}^\tau(\overline{f}) \circ c \\
&= g \circ f_c \circ \mathcal{E}_c^\tau(\text{red}^\tau(\overline{f})) && \text{by Definition 9} \\
&= \phi_c \circ \mathcal{D}_c^\tau(g) \circ \mathcal{E}_c^\tau(\text{red}^\tau(\overline{f})) && \text{by premise} \\
&= \phi_c \circ \mathcal{E}_c^\tau(g \circ \text{red}^\tau(\overline{f})) && \text{by Property 3} \\
&= \phi_c \circ \mathcal{E}_c^\tau(\eta)
\end{aligned}
$$

Thus, by Definition 9, η is equal to $\text{red}^\tau(\overline{\phi})$. $\Box$

If τ is the simple inductive type T, then the unary promotion theorem is identical to the simple promotion theorem for simple reductions, as it is described in [14].

For example, the unary promotion theorem for the simple type $T = \text{bush}(\alpha)$ is:

$$\frac{\begin{aligned}\phi_l(a) &= g(f_l(a)) \\ \phi_b(\text{map}^{list}(g)\,s) &= g(f_b(s))\end{aligned}}{g(\text{red}^{bush}(f_l, f_b)\,x) \ = \ \text{red}^{bush}(\phi_l, \phi_b)\,x}$$

And the unary promotion theorem for the generalized type

$\tau = \text{list}(\alpha) \times \text{list}(\beta)$ is:

$$\frac{\begin{array}{rcl} \phi_{nn}((),()) &=& g(f_{nn}((),())) \\ \phi_{nc}((),(b,s)) &=& g(f_{nc}((),(b,s))) \\ \phi_{cn}((a,l),()) &=& g(f_{cn}((a,l),())) \\ \phi_{cc}(a,(b,g(r))) &=& g(f_{cc}(a,(b,r))) \end{array}}{\begin{array}{l} g(\text{red}^{list\times list}(f_{nn},f_{nc},f_{cn},f_{cc})\ (x,y)) \\ \qquad = \text{red}^{list\times list}(\phi_{nn},\phi_{nc},\phi_{cn},\phi_{cc})\ (x,y) \end{array}}$$

The second promotion theorem is for the case of composing a binary function g with any two n-ary reductions.

Theorem 2 (Binary Promotion Theorem)

$$\frac{\begin{array}{l} \forall c_1 \in \mathcal{GC}(\tau_1), \forall c_2 \in \mathcal{GC}(\tau_2) : \\ \qquad \phi_{c_1\times c_2} \circ \mathcal{M}^{\tau_1\times\tau_2}_{c_1\times c_2}(g) \ = \ g \circ (f_{c_1} \times h_{c_2}) \end{array}}{g \circ (\text{red}^{\tau_1}(\overline{f}) \times \text{red}^{\tau_2}(\overline{h})) \ = \ \text{red}^{\tau_1\times\tau_2}(\overline{\phi})}$$

Proof: Let $F = \text{red}^{\tau_1}(\overline{f})$, $H = \text{red}^{\tau_2}(\overline{h})$, $\eta = g\circ(F\times H)$, and c_1 and c_2 are generalized constructors in $\mathcal{GC}(\tau_1)$ and $\mathcal{GC}(\tau_2)$. Then

$$\begin{array}{rcl} \eta\circ(c_1\times c_2) &=& g\circ(F\times H)\circ(c_1\times c_2) \\ &=& g\circ((F\circ c_1)\times(H\circ c_2)) \\ &=& g\circ((f_{c_1}\circ\mathcal{E}^{\tau_1}_{c_1}(F))\times(h_{c_2}\circ\mathcal{E}^{\tau_2}_{c_2}(H))) \\ && \qquad \text{(by Definition 9)} \\ &=& g\circ(f_{c_1}\times h_{c_2})\circ(\mathcal{E}^{\tau_1}_{c_1}(F)\times\mathcal{E}^{\tau_2}_{c_2}(H)) \\ &=& \phi_{c_1\times c_2}\circ\mathcal{M}^{\tau_1\times\tau_2}_{c_1\times c_2}(g)\circ(\mathcal{E}^{\tau_1}_{c_1}(F)\times\mathcal{E}^{\tau_2}_{c_2}(H)) \\ && \qquad \text{(by premise)} \\ &=& \phi_{c_1\times c_2}\circ\mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(g\circ(F\times H)) \\ && \qquad \text{(by Property 4)} \\ &=& \phi_{c_1\times c_2}\circ\mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(\eta) \end{array}$$

Thus, by Definition 9, η is equal to $\text{red}^{\tau_1\times\tau_2}(\overline{\phi})$. □

For example, for $T_1 = \text{list}(\alpha)$ and $T_2 = \text{nat}$ the binary promotion theorem is:

$$\frac{\begin{array}{rcl} \phi_{nz}((),()) &=& g(f_n(),h_z()) \\ \phi_{ns}((),s) &=& g(f_n(),h_s(s)) \\ \phi_{cz}((a,r),()) &=& g(f_c(a,r),h_z()) \\ \phi_{cs}(a,g(r,s)) &=& g(f_c(a,r),h_s(s)) \end{array}}{\begin{array}{l} g(\text{red}^{list}(f_n,f_c)\ x, \text{red}^{nat}(h_z,h_s)\ y) \\ \qquad = \text{red}^{list\times nat}(\phi_{nz},\phi_{ns},\phi_{cz},\phi_{cs})\ (x,y) \end{array}}$$

4 The Normalization Algorithm

In this section we present our program optimization algorithm. It uses the unary and binary promotion theorems effectively to perform loop fusion. This algorithm, called the *normalization algorithm*, is presented in Figure 1. Term $\mathcal{INV}(g)$ denotes a special intermediate term that should not appear in the normalized term. To enforce this property, the first rule raises an exception if the normalization algorithm encounters such a term. This exception is parameterized by the reduction currently being promoted by a promotion theorem. This parameter guarantees that nested promotion phases will be handled correctly. Normally, $\mathcal{INV}(g)$ is cancelled by g in the *elimination phases* and no exception is raised. If an exception is raised, then it is caught by the undergoing *promotion phases* and no loop fusion is performed. Otherwise, the promotion theorem is used to fuse the two nested reductions into one. Note that the *binary promotion phase* constructs the lambda variables of the new accumulating function by using variable name concatenation $\#(x_1,x_2) = x_1\_x_2$. Variables $\overline{x}$ in the partial reduction and the unary promotion, and $\overline{x_1}$ and $\overline{x_2}$ in the binary promotion phase are vectors of new variable names.

For example, the following is an instance of the unary promotion phase of the normalization algorithm:

$$\mathcal{N}[\![g(\text{red}^{list}(f_n,f_c)\ x)]\!] \quad\rightarrow\quad \text{red}^{list}(\phi_n,\phi_c)\ (\mathcal{N}[\![x]\!])$$

where

$$\begin{array}{rcl} \phi_n &=& \lambda().\mathcal{N}[\![g(f_n())]\!] \\ \phi_c &=& \lambda(a,s).\mathcal{N}[\![g(f_c(a,\mathcal{INV}(g)\ s))]\!] \end{array}$$

The following is an instance of the binary promotion phase of the normalization algorithm:

$$\begin{array}{l} \mathcal{N}[\![g(\text{red}^{list}(f_n,f_c)\ x,\text{red}^{nat}(h_z,h_s)\ y)]\!] \\ \qquad \rightarrow\ \text{red}^{list\times nat}(\phi_{nz},\phi_{ns},\phi_{cz},\phi_{cs})\ (\mathcal{N}[\![x]\!],\mathcal{N}[\![y]\!]) \end{array}$$

where

$$\begin{array}{rcl} \phi_{nz} &=& \lambda((),()).\mathcal{N}[\![g(f_n(),h_z())]\!] \\ \phi_{ns} &=& \lambda((),s).\mathcal{N}[\![g(f_n(),h_s(s))]\!] \\ \phi_{cz} &=& \lambda((a,r),()).\mathcal{N}[\![g(f_c(a,r),h_z())]\!] \\ \phi_{cs} &=& \lambda(a,r\_s).\mathcal{N}[\![g(f_c(a,\mathcal{INV}(g)\ r),h_s(\mathcal{INV}(g)\ s))]\!] \end{array}$$

To prove that the normalization algorithm always terminates, we map any term in our language to the maximum number of nested reductions in the term. That is, for every path in the abstract syntax tree that represents a term we compute the number of reductions in the path and then we take the maximum of all these numbers. It is easy to prove that any recursive call to the normalization algorithm does not increase this number, except in the *combinator reduction* rule. But this rule is applied whenever we have a construction. Since all constructions are finite, this rule always terminates. The only case where the number of nested reductions may remain the same is during the *pair*, the *abstraction*, and the *beta reduction* rules. But these rules always terminate according to the lambda calculus theory.

It is not easy, though, to prove the correctness of the normalization algorithm. In general, we need to define formally the meaning function that maps terms into values and prove that the normalization algorithm always preserves meaning. The only mechanisms we have to prove this are the promotion theorems. We will not present the detailed proof here. Instead we will present a sketch of the proof. The detailed correctness proof for the normalization algorithm which includes only the simple promotion theorem can be found elsewhere [15].

Theorem 3 (Correctness of Normalization) *The normalization algorithm always preserves the meaning of a term.*

Proof sketch: All the transformation rules of the normalization algorithm can be easily proved to preserve the meaning of a term, except for the two promotion laws. We consider the unary promotion phase first. There are two cases for the computation of the new accumulating functions ϕ_c: if any of the computations $\mathcal{N}[\![g(f_c(\mathcal{D}^{\tau}_c(\mathcal{INV}(g))\ \overline{x}))]\!]$ raises the *inverse* exception during the normalization process, then there will be no fusion performed. Otherwise, g is fused with $\mathcal{INV}(g)$ during the normalization process. To see why

$$
\begin{array}{lcll}
\mathcal{N}\llbracket \mathcal{INV}(g) \rrbracket & \rightarrow & \textbf{raise}\ \mathrm{inverse}(g) & \textit{illegal use of } \mathcal{INV}(g) \\
\mathcal{N}\llbracket v \rrbracket & \rightarrow & v & \textit{variable} \\
\mathcal{N}\llbracket c \rrbracket & \rightarrow & c & \textit{construction} \\
\mathcal{N}\llbracket (t_1, t_2) \rrbracket & \rightarrow & (\mathcal{N}\llbracket t_1 \rrbracket, \mathcal{N}\llbracket t_2 \rrbracket) & \textit{pair} \\
\mathcal{N}\llbracket \lambda x.e \rrbracket & \rightarrow & \lambda x.\mathcal{N}\llbracket e \rrbracket & \textit{abstraction} \\
\mathcal{N}\llbracket \mathrm{red}^{\tau}(f_1, \ldots, f_n) \rrbracket & \rightarrow & \mathrm{red}^{\tau}(\mathcal{N}\llbracket f_1 \rrbracket, \ldots, \mathcal{N}\llbracket f_n \rrbracket) & \textit{reduction} \\
\mathcal{N}\llbracket \mathrm{red}^{\tau}(\overline{f})\,(c\,t) \rrbracket & \rightarrow & \mathcal{N}\llbracket f_c(\mathcal{E}^{\tau}_{c}(\mathrm{red}^{\tau}(\overline{f}))\,t) \rrbracket & \textit{combinator reduction} \\
\mathcal{N}\llbracket \mathrm{red}^{\tau_1\times\tau_2}(\overline{f})\,(c\,t_1, t_2) \rrbracket & \rightarrow & \left\{ \begin{array}{l} \textbf{case}\ t_2\ \textbf{of}\ \ldots\ c_i(\overline{x}) \Rightarrow \mathcal{N}\llbracket \mathrm{red}^{\tau_1\times\tau_2}(\overline{f})\,(c\,t_1, c_i(\overline{x})) \rrbracket \ldots \\ \quad \textbf{where}\ c_i \in \mathcal{GC}(\tau_2) \end{array} \right. & \textit{partial reduction} \\
\mathcal{N}\llbracket g(\mathrm{red}^{\tau}(\overline{f})\,t) \rrbracket & \rightarrow & \left\{ \begin{array}{l} \mathrm{red}^{\tau}(\overline{\phi})\,(\mathcal{N}\llbracket t \rrbracket) \\ \quad \textbf{where}\ \forall c \in \mathcal{GC}(\tau): \\ \quad \phi_c = \lambda\overline{x}.\mathcal{N}\llbracket g(f_c(\mathcal{D}^{\tau}_{c}(\mathcal{INV}(g))\,\overline{x})) \rrbracket \\ \textbf{handle}\ \mathrm{inverse}(g) \Rightarrow \mathcal{N}\llbracket g \rrbracket(\mathcal{N}\llbracket \mathrm{red}^{\tau}(\overline{f})\,t \rrbracket) \end{array} \right. & \textit{unary promotion} \\
\mathcal{N}\llbracket g(\mathrm{red}^{\tau_1}(\overline{f})\,t_1, \mathrm{red}^{\tau_2}(\overline{h})\,t_2) \rrbracket & \rightarrow & \left\{ \begin{array}{l} \mathrm{red}^{\tau_1\times\tau_2}(\overline{\phi})(\mathcal{N}\llbracket t_1 \rrbracket, \mathcal{N}\llbracket t_2 \rrbracket) \\ \quad \textbf{where}\ \forall c_1 \in \mathcal{GC}(\tau_1),\ \forall c_2 \in \mathcal{GC}(\tau_2): \\ \quad \phi_{c_1\times c_2} = \lambda(\mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(\#)(\overline{x_1}, \overline{x_2})). \\ \qquad\qquad \mathcal{N}\llbracket g(f_{c_1}(\mathcal{E}^{\tau_1}_{c_1}(\mathcal{INV}(g))\,\overline{x_1}), h_{c_2}(\mathcal{E}^{\tau_2}_{c_2}(\mathcal{INV}(g))\,\overline{x_2})) \rrbracket \\ \textbf{handle}\ \mathrm{inverse}(g) \Rightarrow \mathcal{N}\llbracket g \rrbracket(\mathcal{N}\llbracket \mathrm{red}^{\tau_1}(\overline{f})\,t_1 \rrbracket, \mathcal{N}\llbracket \mathrm{red}^{\tau_2}(\overline{h})\,t_2 \rrbracket) \end{array} \right. & \textit{binary promotion} \\
\mathcal{N}\llbracket g(\mathcal{INV}(g)\,x) \rrbracket & \rightarrow & x & \textit{unary elimination} \\
\mathcal{N}\llbracket g(\mathcal{INV}(g)\,x_1, \mathcal{INV}(g)\,x_2) \rrbracket & \rightarrow & \#(x_1, x_2)\ \equiv\ x_1\_x_2 & \textit{binary elimination} \\
\mathcal{N}\llbracket (\lambda v.e)\,t \rrbracket & \rightarrow & \mathcal{N}\llbracket \mathrm{beta}(v, e, t) \rrbracket & \beta\ \textit{reduction} \\
\mathcal{N}\llbracket f\,e \rrbracket & \rightarrow & \mathcal{N}\llbracket f \rrbracket(\mathcal{N}\llbracket e \rrbracket) & \textit{application}
\end{array}
$$

Figure 1: The Normalization Algorithm

$\mathcal{N}\llbracket g(f_c(\mathcal{D}^{\tau}_{c}(\mathcal{INV}(g))\,\overline{x})) \rrbracket$ computes ϕ_c we use the unary promotion theorem:

$$
\begin{array}{ll}
 & \forall c \in \mathcal{GC}(\tau) :\ \phi_c \circ \mathcal{D}^{\tau}_{c}(g)\ =\ g \circ f_c \\
\Leftrightarrow & \phi_c \circ \mathcal{D}^{\tau}_{c}(g) \circ \mathcal{E}^{\tau}_{c}(\mathcal{INV}(g))\ =\ g \circ f_c \circ \mathcal{E}^{\tau}_{c}(\mathcal{INV}(g)) \\
\Leftrightarrow & \phi_c \circ \mathcal{E}^{\tau}_{c}(g \circ \mathcal{INV}(g))\ =\ g \circ f_c \circ \mathcal{E}^{\tau}_{c}(\mathcal{INV}(g)) \\
 & \qquad \text{(by Property (3))} \\
\Leftrightarrow & \phi_c\ =\ g \circ f_c \circ \mathcal{E}^{\tau}_{c}(\mathcal{INV}(g)) \\
 & \qquad \text{(by unary elimination)}
\end{array}
$$

where $g \circ \mathcal{INV}(g)$ was cancelled out in the unary elimination phase. In order to prove that the binary promotion phase of the normalization algorithm is correct, we use the binary promotion theorem. Let $c_1 \in \mathcal{GC}(\tau_1)$ and $c_2 \in \mathcal{GC}(\tau_2)$. Then from the binary promotion theorem we have:

$$
\begin{array}{ll}
 & \phi_{c_1\times c_2} \circ \mathcal{M}^{\tau_1\times\tau_2}_{c_1\times c_2}(g)\ =\ g \circ (f_{c_1} \times h_{c_2}) \\
\Leftrightarrow & \phi_{c_1\times c_2} \circ \mathcal{M}^{\tau_1\times\tau_2}_{c_1\times c_2}(g) \circ (\mathcal{E}^{\tau_1}_{c_1}(\mathcal{INV}(g)) \times \mathcal{E}^{\tau_2}_{c_2}(\mathcal{INV}(g))) \\
 & \quad =\ g \circ (f_{c_1} \times h_{c_2}) \circ (\mathcal{E}^{\tau_1}_{c_1}(\mathcal{INV}(g)) \times \mathcal{E}^{\tau_2}_{c_2}(\mathcal{INV}(g))) \\
\Leftrightarrow & \phi_{c_1\times c_2} \circ \mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(g \circ (\mathcal{INV}(g) \times \mathcal{INV}(g))) \\
 & \quad =\ g \circ (f_{c_1} \times h_{c_2}) \circ (\mathcal{E}^{\tau_1}_{c_1}(\mathcal{INV}(g)) \times \mathcal{E}^{\tau_2}_{c_2}(\mathcal{INV}(g))) \\
\Leftrightarrow & \phi_{c_1\times c_2} \circ \mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(\#) \\
 & \quad =\ g \circ (f_{c_1} \times h_{c_2}) \circ (\mathcal{E}^{\tau_1}_{c_1}(\mathcal{INV}(g)) \times \mathcal{E}^{\tau_2}_{c_2}(\mathcal{INV}(g))) \quad \square
\end{array}
$$

4.1 Example of a Normalization by Unary Promotion

We will improve $\mathrm{length}(\mathrm{zip}(x, y))$, where:

$$
\begin{array}{lcl}
\mathrm{length}(x) & = & \mathrm{red}^{list}(\lambda().\mathrm{Zero}, \lambda(a, r).\mathrm{Succ}(r))\,x \\
\mathrm{zip}(x, y) & = & \mathrm{red}^{list\times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})\,(x, y) \\
\text{where} & & \left\{ \begin{array}{lcll} f_{nn} & = & \lambda((), ()).\mathrm{Nil} & (u1) \\ f_{nc} & = & \lambda((), (b, s)).\mathrm{Nil} & (u2) \\ f_{cn} & = & \lambda((a, l), ()).\mathrm{Nil} & (u3) \\ f_{cc} & = & \lambda(a, (b, r)).\mathrm{Cons}((a, b), r) & (u4) \end{array} \right.
\end{array}
$$

From the unary promotion phase of the normalization algorithm:

$$
\begin{array}{l}
\mathcal{N}\llbracket \mathrm{length}(\mathrm{red}^{list\times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})\,(x, y)) \rrbracket \\
\qquad \rightarrow\ \mathrm{red}^{list\times list}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc})\,(x, y)
\end{array}
$$

where:

$$\begin{array}{lrcl}
1) & \phi_{nn} & = & \lambda().\text{length}(f_{nn}()) \;=\; \lambda().\text{length}(\text{Nil}) \\
 & & = & \lambda().\text{Zero} \\
2) & \phi_{nc} & = & \lambda((),(b,s)).\text{length}(f_{nc}((),(b,s))) \\
 & & = & \lambda((),(b,s)).\text{length}(\text{Nil}) \;=\; \lambda((),(b,s)).\text{Zero} \\
3) & \phi_{cn} & = & \lambda((a,l),()).\text{length}(f_{nn}((a,l),())) \\
 & & = & \lambda((a,l),()).\text{length}(\text{Nil}) \;=\; \lambda((a,l),()).\text{Zero} \\
4) & \phi_{cc} & = & \lambda(a,(b,r)).\text{length}(f_{cc}(a,(b,\mathcal{INV}(\text{length})(r)))) \\
 & & = & \lambda(a,(b,r)).\text{length}(\text{Cons}((a,b),\mathcal{INV}(\text{length})(r))) \\
 & & & \qquad \text{(by (u4))} \\
 & & = & \lambda(a,(b,r)).\text{Succ}(\text{length}(\mathcal{INV}(\text{length})(r))) \\
 & & & \qquad \text{(by combinator reduction)} \\
 & & = & \lambda(a,(b,r)).\text{Succ}(r) \\
 & & & \qquad \text{(by unary elimination)}
\end{array}$$

Therefore, $\mathcal{N}[\![\text{length}(\text{zip}(x,y))]\!]$ is equal to:

$$\begin{array}{l}
\text{red}^{list\times list}(\lambda().\text{Zero}, \lambda((),(b,s)).\text{Zero}, \\
\qquad \lambda((a,l),()).\text{Zero}, \lambda(a,(b,r)).\text{Succ}(r))(x,y)
\end{array}$$

4.2 Example of a Normalization by Binary Promotion

We will normalize $\text{zip}(\text{map}(f)(x), \text{map}(g)(y))$, which is a binary reduction applied to two unary reductions, where:

$$\text{map}(f)(x) \quad = \quad \text{red}^{list}(\lambda().\text{Nil}, \lambda(a,r).\text{Cons}(f\,a,r))\,x$$

The binary promotion part of the normalization algorithm applied to this case is:

$$\begin{array}{l}
\mathcal{N}[\![\text{zip}(\text{red}^{list}(\lambda().\text{Nil}, \lambda(a,r).\text{Cons}(f\,a,r))\,x, \\
\qquad \text{red}^{list}(\lambda().\text{Nil}, \lambda(b,s).\text{Cons}(g\,b,s))\,y)]\!] \\
\qquad\qquad \rightarrow \; \text{red}^{list\times list}(\phi_{nn},\phi_{nc},\phi_{cn},\phi_{cc})\,(x,y)
\end{array}$$

From the binary promotion phase of the normalization algorithm we have:

$$\begin{array}{lrcl}
1) & \phi_{nn} & = & \lambda((),()).\text{zip}(\text{Nil},\text{Nil}) \;=\; \lambda((),()).\text{Nil} \\
2) & \phi_{nc} & = & \lambda((),(b,s)).\text{zip}(\text{Nil},\text{Cons}(g\,b,s)) \\
 & & = & \lambda((),(b,s)).\text{Nil} \\
3) & \phi_{cn} & = & \lambda((a,r),()).\text{zip}(\text{Cons}(f\,a,r),\text{Nil}) \\
 & & = & \lambda((a,r),()).\text{Nil} \\
4) & \phi_{cc} & = & \lambda(a,(b,r\_s)).\text{zip}(\text{Cons}(f\,a,\mathcal{INV}(\text{zip})\,r), \\
 & & & \qquad \text{Cons}(g\,b,\mathcal{INV}(\text{zip})\,s)) \\
 & & = & \lambda(a,(b,r\_s)).\text{Cons}((f\,a,g\,b), \\
 & & & \qquad \text{zip}(\mathcal{INV}(\text{zip})\,r,\mathcal{INV}(\text{zip})\,s)) \\
 & & = & \lambda(a,(b,r\_s)).\text{Cons}((f\,a,g\,b),r\_s)
\end{array}$$

Therefore, $\text{zip}(\text{map}(f)(x), \text{map}(g)(y))$ is normalized to

$$\begin{array}{l}
\text{red}^{list\times list}(\lambda((),()).\text{Nil}, \lambda((),(b,s)).\text{Nil}, \lambda((a,r),()).\text{Nil}, \\
\qquad \lambda(a,(b,r\_s)).\text{Cons}((f\,a,g\,b),r\_s))(x,y)
\end{array}$$

5 The General Promotion Theorem

In this section we generalize the promotion theorem to capture any combination of n-ary reductions. In order to do that, we generalize the combinator $\mathcal{E}^\tau_c$ as $\mathcal{E}^{\tau/\tau'}_c$:

$$\begin{array}{lcll}
\mathcal{E}^{\tau/\tau'}_c(f) & = & \text{id} & \text{if } \neg\text{inductive}(c) \\
\mathcal{E}^{\tau/T}_c(f) & = & \mathcal{E}^\tau_c(f) & \text{if inductive}(c) \\
\mathcal{E}^{\tau_1\times\tau_2/\tau_1'\times\tau_2'}_{c_1\times c_2}(f_1\times f_2) & = & \mathcal{E}^{\tau_1/\tau_1'}_{c_1}(f_1)\times\mathcal{E}^{\tau_2/\tau_2'}_{c_2}(f_2) &
\end{array}$$

and the combinator $\mathcal{M}^\tau_c$ as $\mathcal{M}^{\tau/\tau'}_c$:

$$\begin{array}{lcll}
\mathcal{M}^{\tau/\tau'}_c & = & \text{id} & \text{if } \neg\text{inductive}(c) \\
\mathcal{M}^{\tau/T}_c(f) & = & \mathcal{D}^\tau_c(f) & \text{if inductive}(c) \\
\mathcal{M}^{\tau_1\times\tau_2/\tau_1'\times\tau_2'}_{c_1\times c_2}(f) & = & & \\
\multicolumn{4}{r}{\lambda(x,y).\mathcal{M}^{\tau_1/\tau_1'}_{c_1}(\lambda z.\mathcal{M}^{\tau_2/\tau_2'}_{c_2}(\lambda w.f(z,w))\,y)\,x}
\end{array}$$

Definition 10 (General Reduction) *A function f of type $\tau\rightarrow\tau'$ is a general reduction (denoted as $\mathcal{G}^{\tau\rightarrow\tau'}$) if it is derived from the following rules:*

$$\begin{array}{ll}
\text{id}^T \in \mathcal{G}^{T\rightarrow T} & \text{where id}^T \text{ is the identity for } T \\
\text{red}^\tau(\overline{f}) \in \mathcal{G}^{\tau\rightarrow T} & \text{if red}^\tau(\overline{f}) \text{ is of type } \tau\rightarrow T \\
f_1\times f_2 \in \mathcal{G}^{(\tau_1\times\tau_2)\rightarrow(\tau_1'\times\tau_2')} & \text{if } f_1\in\mathcal{G}^{\tau_1\rightarrow\tau_1'} \text{ and } f_2\in\mathcal{G}^{\tau_2\rightarrow\tau_2'}
\end{array}$$

For example, length $\times$ id^{nat} is a general reduction from $\mathcal{G}^{(\text{list}\times\text{nat})\rightarrow(\text{nat}\times\text{nat})}$. We will present a promotion theorem for any composition $g\circ f$, where $g:\tau'\rightarrow\alpha$ and $f\in\mathcal{G}^{\tau\rightarrow\tau'}$.

Lemma 1 *For any $f\in\mathcal{G}^{\tau\rightarrow\tau'}$, $g:\tau'\rightarrow\alpha$, and $c\in\mathcal{GC}(\tau)$:*

$$\mathcal{E}^\tau_c(g\circ f) \quad = \quad \mathcal{M}^{\tau/\tau'}_c(g)\circ\mathcal{E}^{\tau/\tau'}_c(f)$$

Proof: If $\tau' = T$ and $f = \text{id}$ or $f = \text{red}^\tau(\overline{f})$ we have $\mathcal{E}^\tau_c(g\circ f) = \mathcal{D}^\tau_c(g)\circ\mathcal{E}^\tau_c(f)$, which is true because of Property 3. We assume the theorem is true for τ_1 and τ_2. For $(f_1\times f_2)\in\mathcal{G}^{(\tau_1\times\tau_2)\rightarrow(\tau_1'\times\tau_2')}$ and $c = c_1\times c_2$ we have:

$$\begin{array}{cl}
\multicolumn{2}{l}{\mathcal{E}^{\tau_1\times\tau_2}_{c_1\times c_2}(g\circ(f_1\times f_2))} \\
= & \lambda(x,y).\mathcal{E}^{\tau_1}_{c_1}(\lambda z.\mathcal{E}^{\tau_2}_{c_2}(\lambda w.g(f_1(z),f_2(w)))\,y)\,x \\
= & \lambda(x,y).\mathcal{M}^{\tau_1/\tau_1'}_{c_1}(\lambda z.\mathcal{M}^{\tau_2/\tau_2'}_{c_2}(\lambda w.g(z,w)) \\
 & \qquad (\mathcal{E}^{\tau_2/\tau_2'}_{c_2}(f_2)\,y))\,(\mathcal{E}^{\tau_1/\tau_1'}_{c_1}(f_1)\,x) \\
= & (\lambda(x,y).\mathcal{M}^{\tau_1/\tau_1'}_{c_1}(\lambda z.\mathcal{M}^{\tau_2/\tau_2'}_{c_2}(\lambda w.g(z,w))\,y)\,x) \\
 & \qquad \circ(\mathcal{E}^{\tau_1/\tau_1'}_{c_1}(f_1)\times\mathcal{E}^{\tau_2/\tau_2'}_{c_2}(f_2)) \\
= & \mathcal{M}^{\tau_1\times\tau_2/\tau_1'\times\tau_2'}_{c_1\times c_2}(g)\circ\mathcal{E}^{\tau_1\times\tau_2/\tau_1'\times\tau_2'}_{c_1\times c_2}(f_1\times f_2) \qquad \Box
\end{array}$$

The combinator $\mathcal{I}_c$, for $f\in\mathcal{G}^{\tau\rightarrow\tau'}$ and $c\in\mathcal{GC}(\tau)$, is defined as follows:

$$\begin{array}{rcl}
\mathcal{I}_c(\text{id}^T) & = & c \\
\mathcal{I}_c(\text{red}^\tau(\overline{f})) & = & f_c \\
\mathcal{I}_{c_1\times c_2}(f_1\times f_2) & = & \mathcal{I}_{c_1}(f_1)\times\mathcal{I}_{c_2}(f_2)
\end{array}$$

Note that the first equation may be derived from the second, since for any inductive type T we have $\text{id} = \text{red}^T(\overline{f})$, where $f_c = c$ for any value constructor c of T.

Lemma 2 *For any $f\in\mathcal{G}^{\tau\rightarrow\tau'}$ and $c\in\mathcal{GC}(\tau)$:*

$$f\circ c \quad = \quad \mathcal{I}_c(f)\circ\mathcal{E}^{\tau/\tau'}_c(f)$$

Proof: If $\tau = \tau' = T$ and $f = \text{id}$ then $\mathcal{I}_c(f)\circ\mathcal{E}^{\tau/\tau'}_c(f) = c\circ E^T_c(f) = c = f\circ c$. For $\tau' = T$ and $f = \text{red}^\tau(\overline{f})$ we have $\mathcal{I}_c(f) = f_c$. Then $f\circ c = f_c\circ\mathcal{E}^\tau_c(f)$, which is the definition of reduction. We assume the theorem is true for τ_1 and τ_2. For $f = (f_1\times f_2)\in\mathcal{G}^{(\tau_1\times\tau_2)\rightarrow(\tau_1'\times\tau_2')}$ and $c = c_1\times c_2$ we have:

$$\begin{array}{rcl}
f\circ c & = & (f_1\times f_2)\circ(c_1\times c_2) \\
 & = & (f_1\circ c_1)\times(f_2\circ c_2) \\
 & = & (\mathcal{I}_{c_1}(f_1)\circ\mathcal{E}^{\tau_1/\tau_1'}_{c_1}(f_1))\times(\mathcal{I}_{c_2}(f_2)\circ\mathcal{E}^{\tau_2/\tau_2'}_{c_2}(f_2)) \\
 & = & (\mathcal{I}_{c_1}(f_1)\times\mathcal{I}_{c_2}(f_2))\circ(\mathcal{E}^{\tau_1/\tau_1'}_{c_1}(f_1)\times\mathcal{E}^{\tau_2/\tau_2'}_{c_2}(f_2)) \\
 & = & \mathcal{I}_c(f)\circ\mathcal{E}^{\tau/\tau'}_c(f) \qquad \Box
\end{array}$$

$$
\begin{array}{lcll}
\mathcal{N}[\![g(f\,t)]\!] & \rightarrow & \left\{\begin{array}{l} \mathrm{red}^{\tau}(\overline{\phi})\,(\mathcal{N}[\![t]\!]) \\ \quad \textbf{where } f \in \mathcal{G}^{\tau\rightarrow\tau'},\ g : \tau' \rightarrow \alpha,\ \text{and } \forall c \in \mathcal{GC}(\tau): \\ \quad \phi_c = \lambda(\mathcal{E}_c^{\tau}(\#)\,\overline{x}).\mathcal{N}[\![g(\mathcal{I}_c(f)(\mathcal{E}_c^{\tau/\tau'}(\mathcal{S}^{\tau}(\mathcal{INV}(g)))\,\overline{x}))]\!] \\ \textbf{handle } \mathrm{inverse}(g) \ \Rightarrow\ \mathcal{N}[\![g]\!](\mathcal{N}[\![f\,t]\!]) \end{array}\right. & \textit{general promotion} \\
\mathcal{N}[\![g(\mathcal{S}^{\tau}(\mathcal{INV}(g))(x_1,\ldots,x_n))]\!] & \rightarrow & \#(x_1,\ldots,x_n) \ \equiv\ x_1\_\ldots\_x_n & \textit{general elimination}
\end{array}
$$

Figure 2: The General Normalization Algorithm

Theorem 4 (General Promotion Theorem)
Let $f \in \mathcal{G}^{\tau\rightarrow\tau'}$ and $g : \tau' \rightarrow \alpha$. Then

$$\frac{\forall c \in \mathcal{GC}(\tau):\ \phi_c \circ \mathcal{M}_c^{\tau/\tau'}(g) \ = \ g \circ \mathcal{I}_c(f)}{g \circ f \ = \ \mathrm{red}^{\tau}(\overline{\phi})}$$

Proof: We will use Lemmas 1 and 2. Let $\eta = g \circ f$ and $c \in \mathcal{GC}(\tau)$. Then

$$
\begin{array}{rcll}
\eta \circ c & = & g \circ f \circ c & \\
& = & g \circ \mathcal{I}_c(f) \circ \mathcal{E}_c^{\tau/\tau'}(f) & \text{by Lemma 2} \\
& = & \phi_c \circ \mathcal{M}_c^{\tau/\tau'}(g) \circ \mathcal{E}_c^{\tau/\tau'}(f) & \text{by premise} \\
& = & \phi_c \circ \mathcal{E}_c^{\tau}(g \circ f) & \text{by Lemma 1} \\
& = & \phi_c \circ \mathcal{E}_c^{\tau}(\eta) &
\end{array}
$$

Thus, η is the reduction $\mathrm{red}^{\tau}(\overline{\phi})$. □

The rules in Figure 2 extend the normalization algorithm, presented in Figure 1, with a general promotion phase. Therefore, the unary and the binary promotion phases as well as the unary and binary elimination phases of this algorithm can be replaced by the rules in Figure 2. In this algorithm, $\mathcal{S}^{\tau}(g)$ creates multiple copies of the function g into a product that has the same shape as τ:

Definition 11 (Combinator $\mathcal{S}$)

$$
\begin{array}{rcl}
\mathcal{S}^{T}(g) & = & g \\
\mathcal{S}^{\tau_1\times\tau_2}(g) & = & \mathcal{S}^{\tau_1}(g) \times \mathcal{S}^{\tau_2}(g)
\end{array}
$$

The correctness proof of this algorithm is derived from the general promotion theorem:

$$
\begin{array}{rl}
& \forall c \in \mathcal{GC}(\tau):\ \phi_c \circ \mathcal{M}_c^{\tau/\tau'}(g) \ = \ g \circ \mathcal{I}_c(f) \\
\Leftrightarrow & \phi_c \circ \mathcal{M}_c^{\tau/\tau'}(g) \circ \mathcal{E}_c^{\tau/\tau'}(\mathcal{S}^{\tau}(\mathcal{INV}(g))) \\
& \qquad = \ g \circ \mathcal{I}_c(f) \circ \mathcal{E}_c^{\tau/\tau'}(\mathcal{S}^{\tau}(\mathcal{INV}(g))) \\
\Leftrightarrow & \phi_c \circ \mathcal{E}_c^{\tau}(g \circ \mathcal{S}^{\tau}(\mathcal{INV}(g))) \ = \ g \circ \mathcal{I}_c(f) \circ \mathcal{E}_c^{\tau/\tau'}(\mathcal{S}^{\tau}(\mathcal{INV}(g))) \\
\Leftrightarrow & \phi_c \circ \mathcal{E}_c^{\tau}(\#) \ = \ g \circ \mathcal{I}_c(f) \circ \mathcal{E}_c^{\tau/\tau'}(\mathcal{S}^{\tau}(\mathcal{INV}(g)))
\end{array}
$$

For example, we will normalize $\mathrm{nth}(d)(\mathrm{append}(x,y),n)$, where $\mathrm{nth}(d)(x,n)$ is:

$$\mathrm{red}^{list\times nat}(\lambda((),()).d, \lambda((),i).d, \lambda((a,l),()).a, \lambda(a,r).r)\,(x,n)$$

We have $f \in \mathcal{G}^{(list\times nat)\mapsto(list\times nat)}$ and
$f = \mathrm{red}^{list}(\lambda().y, \lambda(a,r).\mathrm{Cons}(a,r)) \times \mathrm{id}^{nat}$.

The general promotion part of the normalization algorithm applied to this case is:

$$
\begin{array}{l}
\mathcal{N}[\![\mathrm{nth}(d)(\mathrm{red}^{list}(\lambda().y, \lambda(a,r).\mathrm{Cons}(a,r))\,x, \mathrm{id}^{nat}(n))]\!] \\
\qquad \rightarrow\ \mathrm{red}^{list\times nat}(\phi_{nz}, \phi_{ns}, \phi_{cz}, \phi_{cs})\,(x,n)
\end{array}
$$

where

$$
\begin{array}{llcl}
1) & \phi_{nz} & = & \lambda((),()).\mathrm{nth}(d)(y,\mathrm{Zero}) \\
2) & \phi_{ns} & = & \lambda((),i).\mathrm{nth}(d)(y,\mathrm{Succ}(i)) \\
3) & \phi_{cz} & = & \lambda((a,r),()).\mathrm{nth}(d)(\mathrm{Cons}(a,r),\mathrm{Zero}) \\
& & = & \lambda((a,r),()).a \\
4) & \phi_{cs} & = & \lambda(a,r\_i).\mathrm{nth}(d)(\mathrm{Cons}(a,\mathcal{INV}(\mathrm{nth})\,r), \\
& & & \qquad\qquad \mathrm{Succ}(\mathcal{INV}(\mathrm{nth})\,i)) \\
& & = & \lambda(a,r\_i).\mathrm{nth}(d)(\mathcal{INV}(\mathrm{nth})\,r, \mathcal{INV}(\mathrm{nth})\,i) \\
& & = & \lambda(a,r\_i).r\_i
\end{array}
$$

Therefore, $\mathrm{nth}(d)(\mathrm{append}(x,y),n)$ is

$$
\begin{array}{l}
\mathrm{red}^{list\times nat}(\lambda((),()).\mathrm{nth}(d)(y,\mathrm{Zero}), \\
\qquad \lambda((),i).\mathrm{nth}(d)(y,\mathrm{Succ}(i)), \\
\qquad \lambda((a,r),()).a, \\
\qquad \lambda(a,r\_i).r\_i)\,(x,n)
\end{array}
$$

6 Converting Functional Programs into Algebraic Programs

This section presents an automated method of converting regular recursive function definitions, such as those found in an ML-like language, into algebraic programs with reductions. It uses the normalization algorithm to translate function definitions into general reductions as well as to improve the resulting programs. The normalization algorithm will raise an exception if this translation is impossible. The resulting reductions, though, may be of higher-order. Subsequently, a second phase is needed to convert the higher-order reductions into first-order.

Definition 12 (Recursive Function Definition) *A function $f : ((T_1 \times T_2) \times \cdots \times T_n) \rightarrow S_1 \rightarrow \cdots \rightarrow S_m \rightarrow T'$ has a* recursive function definition *if it can be computed by M number of recursive rules of the form:*

$$f(C_{i_1}(\overline{x_1}),\ldots,C_{i_n}(\overline{x_n}))\,v_1\,\ldots\,v_m \ = \ e_i$$

where v_k are variables and $\overline{x_k}$ are vectors of variables. The left side of this equation must be linear (i.e. every variable should appear only once in the patterns on the left side of the equations).

Before considering the general case of Definition 12 where $n > 1$, we examine the simple case where function f has only one recursive parameter (i.e. where $n = 1$):

Theorem 5 *Let f be a recursive function definition of type $T \rightarrow S_1 \rightarrow \cdots \rightarrow S_m \rightarrow T'$, computed by the recursive rules:*

$$f(C_i(\overline{x}))\,v_1\,\ldots\,v_m \ = \ e_i$$

one for each constructor C_i of T, and let $F = \text{red}^T(\overline{\phi})$ be a unary reduction, where $\forall C_i \in \mathcal{GC}(T)$:

$$\phi_{c_i} \;=\; \lambda\overline{z}.\lambda v_1 \ldots \lambda v_m.\mathcal{N}[\![(\lambda\overline{x}.e_i)\,(E^T_{c_i}(\mathcal{INV}(f))\,\overline{z})]\!]$$

If there is no exception raised during the normalization of a ϕ_{c_i}, then $F = f$.

Proof: $\forall C_i \in \mathcal{GC}(T)$, f satisfies:

$$\begin{array}{rl}
& f \circ C_i \;=\; \lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i \\
\Leftrightarrow & \phi_{c_i} \circ E^T_{c_i}(f) \;=\; \lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i \\
\Leftrightarrow & \phi_{c_i} \circ E^T_{c_i}(f) \circ E^T_{c_i}(\mathcal{INV}(f)) \\
& \qquad = (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ E^T_{c_i}(\mathcal{INV}(f)) \\
\Leftrightarrow & \phi_{c_i} \circ E^T_{c_i}(f \circ \mathcal{INV}(f)) \\
& \qquad = (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ E^T_{c_i}(\mathcal{INV}(f)) \\
\Leftrightarrow & \phi_{c_i} \;=\; (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ E^T_{c_i}(\mathcal{INV}(f)) \qquad \Box
\end{array}$$

For example, list append is defined by the following recursive rules:

$$\begin{array}{lcl}
\text{app}(\text{Nil})\,y & = & y \\
\text{app}(\text{Cons}(a,l))\,y & = & \text{Cons}(a, \text{app}(l)\,y)
\end{array}$$

The inductive program that computes app (generated by the algorithm in Theorem 5) is:

$$\begin{array}{l}
\text{red}^{list}(\lambda().\lambda y.y,\, \lambda(z_1,z_2).\lambda y.\text{Cons}(z_1, \text{app}(\mathcal{INV}(\text{app})(z_2))\,y)) \\
= \text{red}^{list}(\lambda().\lambda y.y,\, \lambda(z_1,z_2).\lambda y.\text{Cons}(z_1, z_2(y)))
\end{array}$$

Note that primitive recursive programs, such as the factorial defined as:

$$\begin{array}{lcl}
\text{fact}(\text{Zero}) & = & \text{Succ}(\text{Zero}) \\
\text{fact}(\text{Succ}(n)) & = & \text{times}(\text{Succ}(n), \text{fact}(n))
\end{array}$$

will fail to produce an inductive program, since the first $\mathcal{INV}(\text{fact})$ in

$$\text{times}(\text{Succ}(\mathcal{INV}(\text{fact})\,n), \text{fact}(\mathcal{INV}(\text{fact})\,n))$$

will not be cancelled out.

We will use the following abbreviations for the general case of Definition 12:

$$\begin{array}{lcl}
\tau & = & ((T_1 \times T_2) \times \cdots \times T_n) \\
c & = & ((C_{i_1} \times C_{i_2}) \times \cdots \times C_{i_n}) \\
\overline{x} & = & ((\overline{x_1}, \overline{x_2}), \ldots, \overline{x_n})
\end{array}$$

In order to generalize Theorem 5 to work on any recursive function definition in Definition 12, we define a new combinator $\mathcal{P}^\tau_c$ to be the product of all functors $\mathcal{E}^{T_i}_{c_i}$, where τ is the product of all T_i and c is the product of all C_i:

Definition 13 (Combinator $\mathcal{P}$)

$$\begin{array}{lcll}
\mathcal{P}^\tau_c(f) & = & \text{id} & \text{if } \neg\text{inductive}(c) \\
\mathcal{P}^T_c(f) & = & E^T_c(f) & \multirow{2}{*}{\text{if inductive}(c)} \\
\mathcal{P}^{\tau_1\times\tau_2}_{c_1\times c_2}(f) & = & \mathcal{P}^{\tau_1}_{c_1}(f) \times \mathcal{P}^{\tau_2}_{c_2}(f) &
\end{array}$$

Property:

$$\mathcal{E}^\tau_c(f) \circ \mathcal{P}^\tau_c(g) \;=\; \mathcal{E}^\tau_c(f \circ \mathcal{S}^\tau(g))$$

where the combinator $\mathcal{S}^\tau$ was defined in Definition 11.

Proof: If c is not inductive then the property is true. Otherwise, if $\tau = T$ then $E^T_c(f) \circ E^T_c(g) = E^T_c(f \circ g)$. If $\tau = \tau_1 \times \tau_2$ and $c = c_1 \times c_2$ then:

$$\begin{array}{l}
\mathcal{E}^\tau_c(f) \circ \mathcal{P}^\tau_c(g) \\
= (\lambda(x,y).\mathcal{E}^{\tau_1}_{c_1}(\lambda z.\mathcal{E}^{\tau_2}_{c_2}(\lambda w.f(z,w))\,y)\,x) \circ (\mathcal{P}^{\tau_1}_{c_1}(g) \times \mathcal{P}^{\tau_2}_{c_2}(g)) \\
= \lambda(x,y).\mathcal{E}^{\tau_1}_{c_1}(\lambda z.\mathcal{E}^{\tau_2}_{c_2}(\lambda w.f(z,w))\,(\mathcal{P}^{\tau_2}_{c_2}(g)\,y))\,(\mathcal{P}^{\tau_1}_{c_1}(g)\,x) \\
= \lambda(x,y).\mathcal{E}^{\tau_1}_{c_1}(\lambda z.\mathcal{E}^{\tau_2}_{c_2}(\lambda w.f(\mathcal{S}^{\tau_1}(g)\,z, \mathcal{S}^{\tau_2}(g)\,w))\,y)\,x \\
= \mathcal{E}^\tau_c(f \circ \mathcal{S}^\tau(g)) \qquad \Box
\end{array}$$

Theorem 6 *Let f be the function computed by the rules in Definition 12 and let $F = \text{red}^\tau(\overline{\phi})$, where for each of the M equations in Definition 12 (i.e. $\forall c \in \mathcal{GC}(\tau)$):*

$$\phi_c \;=\; \lambda(\mathcal{E}^\tau_c(\#)\,\overline{z}).\lambda v_1 \ldots \lambda v_m.\mathcal{N}[\![(\lambda\overline{x}.e_i)\,(\mathcal{P}^\tau_c(\mathcal{INV}(f))\,\overline{z})]\!]$$

If there is no exception raised during the normalization of a ϕ_c, then $F = f$.

Proof: $\forall c \in \mathcal{GC}(\tau)$, f satisfies:

$$\begin{array}{rl}
& f \circ c \;=\; \lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i \\
\Leftrightarrow & \phi_c \circ \mathcal{E}^\tau_c(f) \;=\; \lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i \\
\Leftrightarrow & \phi_c \circ \mathcal{E}^\tau_c(f) \circ \mathcal{P}^\tau_c(\mathcal{INV}(f)) \\
& \qquad = (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ \mathcal{P}^\tau_c(\mathcal{INV}(f)) \\
\Leftrightarrow & \phi_c \circ \mathcal{E}^\tau_c(f \circ \mathcal{S}^\tau(\mathcal{INV}(f))) \\
& \qquad = (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ \mathcal{P}^\tau_c(\mathcal{INV}(f)) \\
\Leftrightarrow & \phi_c \circ \mathcal{E}^\tau_c(\#) \;=\; (\lambda v_1 \ldots \lambda v_m.\lambda\overline{x}.e_i) \circ \mathcal{P}^\tau_c(\mathcal{INV}(f)) \quad \Box
\end{array}$$

For example, the nth element of a list is defined by the following recursive rules:

$$\begin{array}{lcl}
\text{nth}(\text{Nil}, \text{Zero})\,d & = & d \\
\text{nth}(\text{Nil}, \text{Succ}(i))\,d & = & d \\
\text{nth}(\text{Cons}(a,l), \text{Zero})\,d & = & a \\
\text{nth}(\text{Cons}(a,l), \text{Succ}(i))\,d & = & \text{nth}(l,i)\,d
\end{array}$$

The inductive program that computes nth (generated by the algorithm in Theorem 6) is:

$$\begin{array}{l}
\text{red}^{list\times nat}(\lambda((),()).\lambda d.d,\, \lambda((),z_1).\lambda d.d,\, \lambda((z_1,z_2),()).\lambda d.z_1, \\
\qquad \lambda(z_1, z_2\_z_3).\lambda d.\text{nth}(\mathcal{INV}(\text{nth})\,z_2, \mathcal{INV}(\text{nth})\,z_3)\,d) \\
= \text{red}^{list\times nat}(\lambda((),()).\lambda d.d,\, \lambda((),z_1).\lambda d.d, \\
\qquad \lambda((z_1,z_2),()).\lambda d.z_1,\, \lambda(z_1, z_2\_z_3).\lambda d.z_2\_z_3(d))
\end{array}$$

The inductive programs generated by the algorithm in Theorem 6 are higher-order in general, that is, reductions return functions instead of values. If a variable v_k in Theorem 6 appears in some restricted form in every accumulating function ϕ_c, then it can be pulled out from the reduction. The restricted form is as follows: every term $(r\,e_1 \ldots e_m)$ in ϕ_c, where r is an accumulative result variable from $\overline{x}$, has $e_k = v_k$ and no other e_j refer to v_k. Then each term $(r\,e_1 \ldots e_m)$ is converted into $(r\,e_1 \ldots e_{k-1}\,e_{k+1} \ldots e_m)$ and each λv_k is pulled out from the reduction.

For example, app that computes list append was previously converted into:

$$\text{red}^{list}(\lambda().\lambda y.y,\, \lambda(z_1,z_2).\lambda y.\text{Cons}(z_1, z_2(y)))$$

which is simplified under the above algorithm into:

$$\lambda y.\text{red}^{list}(\lambda().y,\, \lambda(z_1,z_2).\text{Cons}(z_1,z_2))$$

Similarly, the nth of a list is simplified into:

$$\begin{array}{l}
\lambda d.\text{red}^{list\times nat}(\lambda((),()).d,\, \lambda((),z_1).d,\, \lambda((z_1,z_2),()).z_1, \\
\qquad \lambda(z_1, z_2\_z_3).z_2\_z_3)
\end{array}$$

An example where this precondition is not met is list reverse $\lambda x.\mathrm{rev}(x)\,\mathrm{Nil}$, where rev is:

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

and the inductive program that computes rev is:

$$\begin{array}{l} \mathrm{red}^{list}(\lambda().\lambda y.y,\lambda(z_1,z_2).\lambda y.\mathrm{rev}(\mathcal{INV}(\mathrm{rev})(z_2))\,(\mathrm{Cons}(z_1,y))) \\ = \mathrm{red}^{list}(\lambda().\lambda y.y,\lambda(z_1,z_2).\lambda y.z_2(\mathrm{Cons}(z_1,y))) \end{array}$$

In this case, λy cannot be pulled out from under the reduction. The resulting program is a second-order reduction that computes list reverse in linear time. Note that variable y in $\mathrm{rev}(x)\,y$ is not an inductive parameter, but instead it serves as an accumulator which is expanded at each inductive step. Programs containing such simple forms of second-order reductions can be normalized by using a promotion theorem similar to the one for first-order reductions [15].

7 Related Work

To our knowledge no other work deals with generic recursion schemes over multiple structures. For simple inductions our work is closely related to Wadler's work on listlessness and deforestation [17, 7, 16] and to Chin's work on fusion [2]. Deforestation works on all first order treeless terms. A treeless term is one which is exactly analogous to a safe term, but is described in a much different manner due to the lack of structure imposed on such terms. Chin 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. The application domain of our fusion algorithm is more restricted than the domain of all these methods, but our algorithm is more effective since it is fully automated. In [8] a new, simple, but very effective, automatic technique is presented for implementing deforestation in a compiler. This method requires that each list-producing functions is expressed as a *build* call and each list-consuming functions is expressed as a *foldr* call. The foldr operator is similar to our list reduction while the build operator is a dual-like function of foldr (this technique is an automation of the *HyloSplit* theorem of Meijer et al. [13]). The technique simply fuses adjacent foldr-build pairs by eliminating them completely $((\mathrm{foldr}\,f)\circ(\mathrm{build}\,g) = g\circ f)$. We believe that this method could be more effective if folds are promoted downwards or builds are promoted upwards in a term until they fuse. This can be achieved effectively by applying promotion theorems.

Our stereotyped recursion schemes as well as the promotion theorems are highly influenced by the Squiggol school of program construction [12, 13, 11, 1]. Their goal is to construct a calculus of programs based on some well-behaved recursion schemes, in which their inductive laws, proved once and for all in their generic form, can be instantiated and used for calculating program transformation as well as for proving properties about programs without the need for discovering new laws or using explicit induction. The promotion theorems are examples of a large class of theorems that come for free. We believe that our notation, which is based on calculus of construction, is more intuitive to functional programmers than their formalism (also called the Bird-Meertens Formalism), which is based on category theory. Even though their work is more general than ours, we provide a fully automated system that use the promotion laws effectively.

8 Conclusion

The functional programming style has been criticized because of its waste of resources caused by the building of intermediate data structures, unused closures, and garbage collection. We believe that this is not caused by the functional style per se, but by the use of unrestricted recursion which makes it difficult to validate the application of well known optimizations in unstructured programs.

Recent language proposals to program with the explicit structure of generic recursion schemes are an attempt to circumnavigate these problems. This paper provides a first step towards automatic optimization and compilation for such languages. To our knowledge the algorithm presented here is the first algorithm (not based upon a fixed set of patterns and datatypes) that automatically performs fusion without a memoization phase, and which deals with multiple inductions.

Acknowledgements: Leonidas Fegaras is supported by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518. Tim Sheard and Tong Zhou are supported in part by a contract with Air Force Material Command (F19628-93-C-0069).

References

[1] R. Bird and O. deMoor. Solving Optimisation Problems with Catamorphisms. In *Mathematics of Program Construction*, pp 45–66. Springer-Verlag, June 1992. LNCS 669.

[2] W. Chin. Safe Fusion of Functional Expressions. *Proceedings of the ACM Symposium on Lisp and Functional Programming, San Francisco, California*, pp 11–20, June 1992.

[3] R. Cockett and T. Fukushima. About Charity. Technical report, Department of Computer Science, the University of Calgary, Alberta, Canada, June 1992. Research Report No. 92/480/18.

[4] L. Fegaras. Efficient Optimization of Iterative Queries. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City*, pp 200–225. Springer-Verlag, Workshops on Computing, August 1993.

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

[6] 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. LNCS 607.

[7] A. Ferguson and P. Wadler. When will Deforestation Stop. In *Proceedings of 1988 Glasgow Workshop on Functional Programming, Rothesay, Isle of Bute*, pp 39–56, August 1988. Also as research report 89/R4 of Glasgow University.

[8] A. Gill, J. Launchbury, and S. Peyton Jones. A Short Cut to Deforestation. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 223–232, June 1993.

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

[10] R. Kieburtz and J. Lewis. Algebraic Design Language (Preliminary Definition). Technical Report #94-002, Oregon Graduate Institute, 1994.

[11] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[12] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, pp 335–347. Springer-Verlag, June 1989. LNCS 375.

[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. LNCS 523.

[14] T. Sheard and L. Fegaras. A Fold for All Seasons. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 233–242, June 1993.

[15] T. Sheard and L. Fegaras. Optimizing Algebraic Programs. Oregon Graduate Institute, Technical report #94-004 A version of this paper is ftp-able from `cse.ogi.edu:/pub/pacsoft/papers/OptAlgProg.ps`.

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

[17] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, pp 344–358, March 1988. LNCS 300.