Optimizing Algebraic Programs

OGI, Tech-report #94-004

Tim Sheard* & Leonidas Fegaras†
Oregon Graduate Institute of Science & Technology
sheard@cse.ogi.edu fegaras@cse.ogi.edu

Abstract

This paper considers a programming language where all control is encoded in algebras and combinators over algebras. This language supports higher levels of abstraction than traditional functional languages and is amenable to calculation based optimization. Three well known transformations are illustrated. Each one, requiring varying levels of insight and creativity over ordinary functional programs, can be fully automated in an algebraic language. The algorithm encoding these transformations is presented. This algorithm is an improvement over our previous work since it works over a richer, more expressive language, encodes more transformations, and is more efficient.

1 Introduction

We have developed a programming style we call *algebraic programming* because of its reliance on algebras and combinators for encoding control. Algebraic programs provide two advantages over traditional functional programs. First, they provide a mechanism for abstracting over type constructors, i.e. it is possible to encode algorithms which work over *any* datatype definition. Second, such algorithms have generic theorems, which make it possible to build semantic based optimizations. Such optimizations are amenable to automation, whereas languages which allow arbitrary recursive programs lack the structure necessary for this kind of automation without expensive analysis.

The contributions of this paper are several. First we extend our earlier work[5, 6, 17] by embedding our optimizations in a richer language, and describe an improved algorithm for computing them. Our previous work focused on a restricted language which has now been extended to include all the features of a modern functional programming language. The improved algorithm works over the entire extended language while the original algorithm worked only on a syntactically identifiable subset of the restricted language.

*Tim Sheard is supported in part by a contract with Air Force Material Command (F19628-93-C-0069).
†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.

Second, we report on an actual implementation[12] which includes a user oriented front-end as well as our optimizing back-end, and a compiler. Experiments with our implementation illustrate that is feasible, practical, and beneficial to program in an algebraic style.

Third, we illustrate that it is possible to extend our techniques by the use of optimization algorithms based upon additional generic theorems. We describe three transformations which heretofore either required human intervention in the form of "eureka" steps or generalization choices, or time consuming search based analysis. The explicit structure of algebraic programs makes these transformations evident by simple inspection or reduction based techniques. In fact, we have developed and implemented an algorithm which utilizes just these three transformations and which manages to capture many well known optimizations. It is our belief that additional generic strategies can strengthen our algorithm.

2 Three Transformations

In the this section we quickly outline three transformations in terms of traditional functional programming. The first two were first described over traditional functional programs by Burstall and Darlington[1], the third is a generic construction of distributive laws we believe is new in any context. In subsequent sections we describe how algebraic programs make these transformations evident.

2.1 Simultaneous Traversal of a Single Structure

Two algorithms which both traverse a single structure can often be fused into a single traversal. This technique is a generalization of loop fusion to arbitrary data structures. As an example consider computing the length and sum of a list simultaneously as in the example $avg\ x = (sum\ x)/(len\ x)$. Given the definitions

$$\begin{array}{lcllcl} len\ \mathrm{nil} & = & 0 & sum\ \mathrm{nil} & = & 0 \\ len\ (\mathrm{cons}(x, xs)) & = & 1 + (len\ xs) & sum\ (\mathrm{cons}(x, xs)) & = & x + (sum\ xs) \end{array}$$

we proceed by introducing a function that computes both values: $h\ x = (len\ x,\ sum\ x)$, and by instantiating this function on the two cases which construct legal lists

$$\begin{array}{lcl} h\ nil & = & (len\ 0, sum\ 0) \\ & = & (0, 0) \\ \\ h\ (\mathrm{cons}(x, xs)) & = & (len(\mathrm{cons}(x, xs)), sum(\mathrm{cons}(x, xs))) \\ & = & (1 + len\ xs, x + sum\ xs) \end{array}$$

At this point we need to generalize two of the terms in order to complete the transformation. It is this step that often requires human intervention.

$$\begin{array}{lcl} h\ (\mathrm{cons}(x, xs)) & = & (1 + u, x + v)\ \textbf{where}\ (u, v) = (len\ xs, sum\ xs) \\ & = & (1 + u, x + v)\ \textbf{where}\ (u, v) = h\ xs \end{array}$$

Thus completing the definition of the function that simultaneously computes both the length and sum of a list.

2.2 Deforestation

Functional programming encourages the definition of complex functions as the composition of simpler ones. This often introduces an unnecessary intermediate data structure. For example consider computing: $len(map\ f\ x)$ where

$$\begin{array}{lcl} map\ f\ \text{nil} & = & \text{nil} \\ map\ f\ (\text{cons}(x, xs)) & = & \text{cons}(f\ x,\ map\ f\ xs) \end{array}$$

Computing $(map\ f\ x)$ produces an intermediate list which is then consumed by len. We proceed in much the same manner as for simultaneous traversal, introducing a new function h, and using the definitions to "arrange" a *recursive* call to h.

$$\begin{array}{lcl} h\ x & = & len(map\ f\ x) \\ & & \\ h\ \text{nil} & = & len(map\ f\ \text{nil}) \\ & = & len(\text{nil}) \\ & = & 0 \\ & & \\ h\ (\text{cons}(x, xs)) & = & len(map\ f\ (\text{cons}(x, xs))) \\ & = & len(\text{cons}(f\ x,\ map\ f\ xs)) \\ & = & 1 + (len(map\ f\ xs)) \\ & = & 1 + h\ xs \end{array}$$

In this example "arranging" for the recursive call to appear was easy. In general this is not the case.

2.3 Distributive Law Generation

Arranging for the recursive call to appear often depends upon knowledge of distributive laws. For example consider $h\ x\ =\ len(rev\ x)$ where

$$\begin{array}{lcl@{\qquad}lcl} \text{nil}\ @\ y & = & y & rev\ \text{nil} & = & \text{nil} \\ (\text{cons}(x, xs))\ @\ y & = & \text{cons}(x,\ xs\ @\ y) & rev\ (\text{cons}(x, xs)) & = & (rev\ xs)\ @\ (\text{cons}(x, \text{nil})) \end{array}$$

For the case $x\ =\ \text{cons}(y, ys)$ we proceed

$$\begin{array}{lcl} h\ (\text{cons}(y, ys)) & = & len(rev\ (\text{cons}(y, ys))) \\ & = & len((rev\ ys)\ @\ (\text{cons}(y, \text{nil}))) \end{array}$$

At this point we need the distributive law: $len(x\ @\ y)\ =\ (len\ x) + (len\ y)$ to complete the transformation.

$$\begin{array}{cl} = & len(rev\ ys)\ +\ len(\text{cons}(y, \text{nil})) \\ = & len(rev\ ys)\ +\ 1\ +\ len(\text{nil}) \\ = & len(rev\ ys)\ +\ 1\ +\ 0 \\ = & h(ys)\ +\ 1 \end{array}$$

Algebraic programs provide the means to generate this law (and many others) on demand, thus somewhat alleviating the need for a library of laws, which no matter how large, will often be incomplete.

3 Algebraic Programming in ADL

In this section we describe our implementation of algebraic programming we call ADL (*Algebraic Design Language*).

3.1 Signatures

ADL departs significantly from functional programming languages such as SML by providing declarations of signatures that define varieties of structure algebras, not simply datatypes. An algebraic signature consists of a finite set of operator names, together with the type of the domain of each operator. The co-domain of an operator is the carrier type for each particular algebra. For example:

```
signature List{type c; $list(a)/c = {$nil, $cons of (a * c)}}
signature Tree{type c; $tree(a)/c = {$tip, $fork of c * a * c}}
```

The *list*-sorted algebras have a signature parametric on a type represented by the variable a. The type variable c is used as the name of the carrier type. The signature consists of a pair of operator names, with typing: $\$nil : c, \$cons : a \times c \rightarrow c$. Note that the domain of each operator is explicit in the signature, and the codomain of each operator is implicitly given by the carrier.

Defining a signature in ADL causes several types and functions to be defined automatically. One is the *freely constructed* type which would correspond to a datatype definition in language like ML. Several other functions and types are defined automatically as well. The induced types and functions of the list signature would be declared as follows in Standard ML:

```
datatype 'a list = nil | cons of ('a *  'a list);
datatype ('a,'c) E$list = $nil | $cons of ('a * 'c);
fun E$nil  (f_a,f_c) () = ();
fun E$cons (f_a,f_c) (x,y) = (f_a x, f_c y);
```

All the functions and types above become accessible to the programmer by simply writing the signature declaration. Note that `list` is the usual, recursive, freely constructed type, where the carrier `c` has been replaced with the type being defined (`'a list`), and `cons` and `nil` are the free constructors. The type `E$list` is a non-recursive type which has one extra type parameter corresponding to the carrier of the signature. Constructors of this non-recursive type have the same name as the signature operators, e.g. `$cons, $nil` (different from the free constructors `cons` and `nil`) and are called *algebra operators*. To construct the definitions of the functions `E$nil` and `E$cons` from a signature declaration we are guided by the types of the domains of the corresponding signature operators `$nil` and `$cons`. This can be done by lifting the `*` operator to work on functions as follows: `(f * g)(x,y) = (f x, g y)`. This construction guarantees that the functions `E$nil` and `E$cons` are functors. Like any functor they preserve identities and compositions:

$$\begin{array}{lcl} E\$cons(id, id) & = & id \\ E\$cons(f,g) \circ E\$cons(h,k) & = & E\$cons(f \circ h, g \circ k) \end{array}$$

These functions will play an important part in the sequel. Similar types and functions are induced for `tree` and other signatures.

3.2 Algebras

A concrete algebra is specified by a structure that contains bindings for the carrier type and for each operator of the algebra. Examples for the `list` and `tree` algebras include:

```
List{c := int; $list{$nil := 0, $cons := \(x,y) 1+y}}
List{c := int; $list{$nil := 0, $cons := \(x,y) x+y}}
Tree{c := list(a); $tree{$tip := nil, $fork := \(x,y,z) x @ [y] @ z}}
```

Given the type of the carrier, the assignment of functions to the operators of the signature must be consistently typed.

3.3 Combinators

ADL encodes control in a standard set of combinators that take an algebra as an argument and return a function which is a free algebra morphism (a function over the freely constructed type). The combinators include *map*, *reduction* (fold,catamorphism), *primitive recursion* (paramorphism), *derive* (unfold, anamorphism), and *hom* (hylomorphism), as well as the duals of these combinators (*cohom* etc.) and mechanism for interpreting all these morphisms in an arbitrary monad. We will not discuss these additional mechanisms here[12].

When a combinator is applied to an algebra specification the returned morphism obeys a set of recursive equations particular to that algebra. For example the `red` combinator applied to a `list` algebra obeys:

$$\begin{array}{lcl} \mathrm{red}[\mathit{list}]\ \mathit{List}\{c; \mathit{\$nil}, \mathit{\$cons}\}\ \ \mathrm{nil} & = & \mathit{\$nil} \\ \mathrm{red}[\mathit{list}]\ \mathit{List}\{c; \mathit{\$nil}, \mathit{\$cons}\}\ \ (\mathrm{cons}\,(x,y)) & = & \mathit{\$cons}(x,\, \mathrm{red}[\mathit{list}]\ \mathit{List}\{c; \mathit{\$nil}, \mathit{\$cons}\}\ ) \end{array}$$

Note that a combinator cannot be typed in an ML-like language since it is parametric over algebras of any signature*. In general, the recursive equation a combinator obeys can only be described in a meta-language (since ADL itself has no recursion). In the meta-language this can be done using the induced functors, $E\$c_i$. Given any constructor c_i with type $t_i \to T$ the reduction combinator obeys:

$$\mathrm{red}[T]\ \ T\{\$c_i := f_i\}\ \ (c_i\ x)\ \ =\ \ f_i\ (E\$c_i\ (id, \mathrm{red}[T]\ \ T\{\$c_i := f_i\})\ x)$$

Some example definitions in ADL of functions using `red` are:

*In ML we could represent an algebra by a tuple of functions. A single function `red` that takes any algebra as input and returns a reduction for that algebra cannot be typed.

```
val sum = red[list] List{c:=int; $nil := 0, $cons := (op +) };
val len = red[list] List{c:=int; $nil := 0, $cons := \(x,y) y+1 };
val map f = red[list] List{c:=list(a); $nil := nil, $cons := \(x,y) cons(f x,y) };
val flatten = red[tree] Tree{c:=list(a); $tip:=nil, $fork:=\(l,x,r) l @ [x] @ r};
```

The `hom` combinator applied to a *list* algebra with carrier c and a splitting function $P : \gamma \to E\$list(\alpha, \beta)$ returns a morphism with type $\gamma \to c$ which obeys the recursive equation:

$$
\begin{array}{l}
\mathrm{hom}[list]\ List\{c;\ \$nil := f,\ \$cons := g\}\ P\ x\ \ = \\
\textbf{case}\ P\ x\ \textbf{of} \\
\quad \$nil \Rightarrow f \\
\ \ \mid \$cons\ (a, b) \Rightarrow g(a,\ \mathrm{hom}[list]\ List\{c;\ \$nil := f,\ \$cons := g\}\ P\ b)
\end{array}
$$

The function $(hom[T]\ T\{\ldots\}\ P\ x)$ recurses over the structure of T found in x which is induced by applying the splitting function P. The general equation can again be expressed in terms of the induced functors.

$$
\begin{array}{l}
\mathrm{hom}[T]\ \ T\{\ldots, \$c_i := f_i, \ldots\}\ P\ (c_i\ x)\ \ = \\
\textbf{case}\ P\ x\ \textbf{of} \\
\quad \ldots \\
\ \ \mid \$c_i\ y \Rightarrow f_i(E\$c_i\ (id, \mathrm{hom}[T]\ \ T\{\ldots, \$c_i := f_i, \ldots\}\ P)\ y) \\
\quad \ldots
\end{array}
$$

An example definition in ADL of a function using `hom` is the `upto` function.

```
val upto = \(m) \(n) (hom[List] {c:= List(int); $nil := nil, $cons := cons}
                          (\ x (if x>n then $nil else $cons(x,x + 1))) m );
```

A slightly more complicated example is quick-sort

```
val flatAlg = {c:=list(a); $tip := nil, $fork := \(l,x,r) l @ [x] @ r};
val Split = \(x) case x of
                   nil => $tip
                 | cons(x,y) => $fork(filter (<= x) y,x,filter (> x) y);
val quicksort = hom[tree] flatAlg Split;
```

In a sense the `hom` combinator is more general than the `red` and other combinators, since all of the other combinators can be expressed in terms of `hom`. For example

$$
\begin{array}{rcl}
\mathrm{red}[T]\ \ T\{\ldots, \$c_i := f_i, \ldots\} & = & \mathrm{hom}[T]\ \ T\{\ldots, \$c_i := f_i, \ldots\}\ out^T \\
 & & \textbf{where:}\ out^T \circ C_i\ =\ \$C_i
\end{array}
$$

Out^T is a particularly simple splitting function, it replaces the top-most free constructor with the its corresponding algebra operator from the type $E\$T$. This has particular importance for our transformation techniques, since the internal data structures representing programs manipulated by our algorithms need deal with only a single combinator. The user may use the simpler combinators which are translated by the compiler into `hom`.

4 The Promotion Theorem.

The promotion theorem for `red` describes the conditions under which the composition of a function g with a `red` can be expressed as another `red` [14, 15]. For a list algebra the theorem is given below.

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

A similar theorem can be expressed for every signature. In general the promotion theorem can be given for `hom` as well as `red` and can be expressed in terms of the induced functors $E\$c_i$ for any signature:

$$\frac{\forall i: \ h_i \circ E\$c_i(id, g) \ = \ g \circ f_i}{g \circ (\text{hom}[T]\{\ldots \$c_i := f_i \ldots\}\ P) \ = \ \text{hom}[T]\{\ldots \$c_i := h_i \ldots\}\ P}$$

The promotion theorem, when used as a left to right rewrite rule, implements a form of fusion. To apply it, the functions, h_i, which meet the stated conditions of the premise must be found.

5 Three transformations for Algebraic Programs

In this section we describe how the structure of algebraic programs enables the three transformations described earlier.

5.1 Deforestation and The Normalization Algorithm

The Normalization algorithm is an effective algorithm for computing the h_i's of the promotion theorem. It based upon the algorithm of [17] extended in the following ways. First, it is based upon the `hom` promotion theorem rather than `red`, second it computes over a richer language, and terminates over the complete language rather than a syntactically identifiable subset. This algorithm automates deforestation [8, 20, 19] and fusion [2] for algebraic programs.

From the promotion theorems we know *only the property* that the h_i's should obey, not how to compute them. The following construction is the basis for the normalization algorithm. Given:

$$h_i \circ E\$c_i(id, g) \quad = \quad g \circ f_i \quad \text{the requisite property}$$

suppose there exists a function g^{-1} with property $g(g^{-1}\ x) \ = \ x$. Then †:

$$\begin{array}{lcll} h_i \circ E\$c_i(id, g) \circ E\$c_i(id, g^{-1}) & = & g \circ f_i \circ E\$c_i(id, g^{-1}) & \\ h_i \circ E\$c_i(id, g \circ g^{-1}) & = & g \circ f_i \circ E\$c_i(id, g^{-1}) & \text{by functorality of } E\$c_i \\ h_i \circ E\$c_i(id, id) & = & g \circ f_i \circ E\$c_i(id, g^{-1}) & \text{by property of } g^{-1} \\ h_i & = & g \circ f_i \circ E\$c_i(id, g^{-1}) & \text{by functorality of } E\$c_i \end{array}$$

†Since in reality g^{-1} may not exist, in the algorithm it only plays the role of a placeholder as shall see in the sequel.

The normalization algorithm works on the formula $g \circ f_i \circ E\$c_i(id, g^{-1})$; it attempts to push the g towards the g^{-1} so that they may cancel each other. In order for the algorithm to be effective, it must rely on no property of g^{-1} other than $g(g^{-1}\ x) = x$, and remove all occurrences of g^{-1}. For example consider

```
len(map f x) = len(red[list] {c:=list(a); $nil:=nil, $cons:=\(x,y) cons(f x,y)})
```

Setting up the equation for `nil` we begin:

```
$nil = len nil = 0
```

for `cons` we proceed:

```
$cons(z,zs) = len ( (\(x,y) cons(f x,y)) (E$cons(id,Inv(len)) (z,zs)) )
$cons(z,zs) = len ( (\(x,y) cons(f x,y)) (z, (Inv(len)) zs)) )
$cons(z,zs) = len ( cons(f z, (Inv(len)) zs) )
$cons(z,zs) = 1 + (len ((Inv(len)) zs) )
$cons(z,zs) = 1 + zs
```

The normalization algorithm is a reduction engine which carries out this process and which instantly recognizes illegal uses of g^{-1} by using exceptions. Its reduction rules are based upon β-reduction, reduction of combinators over freely constructed objects, and the promotion theorem. Given an inverse free term, it returns an equivalent term (possibly the same term). For simplicity the algorithm here is expressed in terms of `red`, though our actual implementation is based upon `hom`.

$N\ term = \textbf{case}\ term\ \textbf{of}$

g^{-1}	$\Rightarrow$	$\textbf{raise}\ inverse$	illegal use of inverse
v	$\Rightarrow$	v	variable
$(t_1, \ldots, t_n)$	$\Rightarrow$	$((N\ t_1), \ldots, (N\ t_n))$	tuple
$\lambda\, v\,.\, e$	$\Rightarrow$	$\lambda\, v\,.\,(N\ e)$	abstraction
$c_i\ \bar{x}$	$\Rightarrow$	$c_i\ (N\ \bar{x})$	construction
$(\lambda\, v\,.\, b)\ x$	$\Rightarrow$	$N(\text{Beta}\ v\ b\ x)$	β reduction
$g(g^{-1}y)$	$\Rightarrow$	y	Success !!!
$g(\text{red}[T]\{\$c_i := f_i\}\ x)$	$\Rightarrow$	$\left\{\begin{array}{l}(\text{red}[T]\{\$c_i := h_i\}\ x) \\ \quad \text{where } h_i\ = \lambda\,\bar{y}\,.\,N(g(f_i(E\$c_i\ (id, g^{-1})\ \bar{y}))) \\ \textbf{handle}\ inverse \Rightarrow \\ \quad (Ng)(N(\text{red}[T]\{\$c_i := f_i\}\ x))\end{array}\right.$	promotion theorem
$f\ x$	$\Rightarrow$	$(N\ f)\ (N\ x)$	normal application
$\text{red}[T]\{\$c_i := f_i\}\ (c_i\ x)$	$\Rightarrow$	$N(f_i(E\$c_i(id, \text{red}[T]\{\$c_i := f_i\})\ x))$	combinator reduction
$\text{red}[T]\{\$c_i := f_i\}\ x$	$\Rightarrow$	$\text{red}[T]\{\$c_i := (N\ f_i)\}\ (N\ x)$	

Where the variables $\bar{y}$ introduced in the promotion step are new variables. If in the promotion step, N fails to compute the h_i, this is signaled by the exception *inverse* and handled by normalizing the two pieces of the promotion step independently, thereby no longer introducing any inverse terms. A proof of correctness of the normalization algorithm can be found in a technical report[18].

5.2 Simultaneous Traversal and the Tupling Lemma

Algebraic programs make it easy to recognize situations where simultaneous traversal is applicable. *Any two* reductions over the *same* variable can be traversed simultaneously! For example consider the example:

$$
\begin{array}{l}
(len\ x,\ sum\ x)\ = \\
(\mathrm{red}[list]\{\$nil := 0, \$cons := \lambda\,(x,y)\,.\,1+y\}\ x,\ \mathrm{red}[list]\{\$nil := 0, \$cons := \lambda\,(x,y)\,.\,x+y\}\ x)
\end{array}
$$

This example transforms into

$$\mathrm{red}[list]\{\$nil := (0,0), \$cons := \lambda\,(x,(u,v))\,.\,(1+u, x+v)\}\ x$$

A general formula for this transformation is called the Tupling lemma[10]. It can be stated for `red` as follows:

$$
\begin{array}{l}
(\ \mathrm{red}[T]\{\ldots, \$c_i := f_i, \ldots\}\ x\ ,\ \mathrm{red}[T]\{\ldots, \$c_i := g_i, \ldots\}\ x\ )\ = \\
\ \mathrm{red}[T]\{\ldots, \langle\ (f_i \circ (E\$c_i(id, \mathit{first})), (g_i \circ (E\$c_i(id, \mathit{second}))\ \rangle, \ldots\}\ x
\end{array}
$$

where $\langle f, g\rangle\ x\ =\ (f\ x,\ g\ x)$ and $\mathit{first}(x,y)\ =\ x$ and $\mathit{second}(x,y)\ =\ y$. An analogous formula for `hom` is also useful.

$$
\begin{array}{l}
(\ \mathrm{hom}[T]\{\ldots, \$c_i := f_i, \ldots\}\ P\ x\ ,\ \mathrm{hom}[T]\{\ldots, \$c_i := g_i, \ldots\}\ P\ x\ )\ = \\
\ \mathrm{hom}[T]\{\ldots, \langle\ (f_i \circ (E\$c_i(id, \mathit{first})), (g_i \circ (E\$c_i(id, \mathit{second}))\ \rangle, \ldots\}\ P\ x
\end{array}
$$

The explicit nature of control in algebraic programs make it possible to recognize and combine two traversals over a single data structure into a single traversal by inspection.

5.3 Generating Distribution Laws for Zero Replacements

Chin [2] relates how the use of laws may improve the deforestation process. We illustrated this in Section 2.3. The explicit structure of algebraic programs makes it possible to calculate some of the necessary laws on demand. In this section we describe how this may be done for a large class of programs. We believe that similar constructions can be found for other classes as well. A common function over an arbitrary type T which has a unique zero constructor, Z, (a nullary constructor like *nil* for list) is defined by:

$$(Zr^T\ y)\ x\ =\ \mathrm{red}\,[T]\,\{T;\ \ldots, \$Z = y,\ \ldots\ , \$C_i = C_i,\ \ldots\}\ x$$

Here the operator for the nullary constructor is assigned y as its meaning, and every other operator is assigned its corresponding free constructor. We call this function a *zero replacement function.* Zr^T has type $T \rightarrow T \rightarrow T$. A function $h(x,y) = Zr\ y\ x$ is associative ($h(w, h(x,y))\ =\ h(h(w,x),y)$), and has the zero, Z, for both a left and right identity ($h(x,Z)\ =\ x$ and $h(Z,y)\ =\ y$) [13]. Recognize that Zr^{list} is the list append operator, and that Zr^{nat} is natural number addition.

We postulate that zero replacement functions have an additional important property: $\forall f \epsilon\ \mathrm{red}[T],\ \exists g\ :\ f \circ (Zr\ y)\ =\ g \circ f$. We make this postulation since the Normalization algorithm provides an effective method to compute g.

For example consider the term $length(x\ @\ y)$. Both $length$ and @ are reductions so there must be a function g such that $length(x\ @\ y)\ =\ g(length\ x)$[‡]. Since $length$ is a reduction suppose that g is a reduction as well and can be expressed as $\mathrm{red}[nat]\ \{\$Zero := \mathbf{m}, \$Succ := \mathbf{n}\}$, where $\mathbf{m}$ and $\mathbf{n}$ are arbitrary unknown functions. By normalizing both $length(x\ @\ y)$ and $\mathrm{red}[nat]\ \{\$Zero := \mathbf{m}, \$Succ := \mathbf{n}\}\ (length\ x)$ we obtain two reductionss which compute the same value. Matching these terms against each other we find bindings for $\mathbf{m}$ and $\mathbf{n}$, thus effectively computing g. For example, $length\ (x\ @\ y)$ normalizes to:

$$
\begin{array}{l}
\mathrm{red}\ \ [list] \\
\qquad \{\$nil\ :=\ \boxed{\mathrm{red}[list]\ \{\$nil\ :=\ Zero,\ \$cons\ :=\ \lambda\,(y0, y1)\,.\,(Succ\, y1)\ \}\ y}, \\
\qquad\ \ \$cons\ :=\ \lambda\,(y3, y2)\,.\,(\boxed{Succ}\ y2)\}
\end{array}
$$

And the term $g(length\, x)\ =\ \mathrm{red}[nat]\{\$Zero\ :=\ m,\ \$Succ\ :=\ n\}\ (length\, x)\}$ normalizes to:

$$
\begin{array}{l}
\mathrm{red}\ \ [list] \\
\qquad \{\$nil\ :=\ \boxed{\mathbf{m}}, \\
\qquad\ \ \$cons\ :=\ \lambda\,(y4, y5)\,.\,(\boxed{\mathbf{n}}\ y5)\}\ x
\end{array}
$$

Matching[§] the two terms we find bindings for $\mathbf{m}$ and $\mathbf{n}$. This is illustrated by the boxed terms in the diagram above. Recognize that $\mathbf{m}$ computes $length\ y$ thus g is $\mathrm{red}[list]\ \{\$nil\ := length\ y,\ \$cons\ :=\ \lambda\,(x, y)\,.\,Succ\, y\}$ which can be recognized as $g\ =\ \lambda\, x\,.(length\ y) + x$. Thus we have effectively computed the law $length(x\ @\ y)\ =\ g(length\ x)\ =\ (length\ y) + (length\ x)$.

This technique allows us to calculate such laws as

$$
\begin{array}{lcl}
map\ f\ (x\ @\ y) & = & (map\ f\ x)\ @\ (map\ f\ y) \\
length\,(x\ @\ y) & = & (length\, x)\ +\ (length\, y) \\
rev\,(x\ @\ y) & = & (rev\, x)\ @\ (rev\, y) \\
w * (x\ +\ y) & = & (w * x)\ +\ (w * y)
\end{array}
$$

This makes possible the automatic fusion of unsafe terms[2], such as $map\ f\ (rev\, x)$ without the need of any additional laws as was illustrated in Section 2.3 above.

6 Applicability of Algebraic Programs and Future Work

The wide spread use of algebraic programming will depend upon several factors. First the ease of expressing programs algebraically, second the generality of the optimization techniques presented here to typical programs, and third the expressiveness of algebraic programs.

Our experience programming algebraically is that it is no harder to program using combinators, than it is using recursion. In fact for some applications it is easier since the combinators

‡Note that g is probably dependent on y.

§Unification where variables, $\mathbf{m}$ and $\mathbf{n}$ in this case, may appear in only one of the terms.

encode typical control patterns and relieve the programmers of tedious detail. Unusual control patterns often must be cast as co-combinators and our experience here is more limited. For some applications, algorithms can actually be constructed which are independent of their data structures. This has important implications for reusability.

The optimizations techniques are widely applicable, and we are currently investigating several other techniques which would make them even more so. Several generic theorems about combinators which return functions (rather than values) are currently being investigated. We feel that this is a particularly valuable avenue to pursue since program which deal with state are quite common and can be cast algebraically using this technique.

The language presented in this paper is limited in that algebraic programs can only traverse a single data structure at a time. This makes algorithms as simple natural number subtraction resort to arcane tricks or be simply unencodable. We would like to encode simple algorithms such as equality, unification, zip, and the nth element of a list function algebraically, and for these functions to be amenable to automatic transformations. Elsewhere[7] we report initial results on encoding these functions algebraically by generalizing the $E\$c_i$ functors and the combinators. These generalizations have promotion like theorems but while our experience with generic transformations is limited, our results so far have been quite encouraging.

7 Related Work

This work is related to Water's on series expressions [21]. His techniques apply only to traversals of linear data structures such as lists, vectors, and streams.

It is also related to Wadler's work on listlessness, and deforestation [19, 20, 8]. Deforestation works on all first order treeless terms. Treelessness is a syntactic property which guarantees that terms can be unfolded without introducing infinite regress. Wadler makes the observation that some intermediate data structures for primitive types, such as integers, booleans, etc. do not really take up space, so he developed a method to handle such terms, using a technique he calls blazing which extends the class of treeless programs. Treelessness can be applied to algebraic programs and normalizing a treeless program is one of the reasons the *inverse* exception would be raised. This is handled by the normalization algorithm by essentially skipping over the offending term. Wadlers language may encode algorithms which induct over several objects simultaneously which cannot be handled by the algebraic language described here.

Chin's work on fusion [2] 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 in much the same manner the normalization algorithm does. 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, and invents laws on the fly that Chin's algorithm must know a priori.

Gill, Launchbury and Jones[11] describe a deforestation algorithm currently used in the

Glasgow Haskell compiler. It uses two combinators *fold*¶ and *build*‖, and a higher order theorem which relates the composition of the two. These techniques are limited since the two forms must be immediately adjacent, while the normalization algorithm will attempt to push these forms through intermediate compositions. In addition functions defined in terms of these combinators are hardwired into the compiler and there is no ability for a user to construct his own, other than through composition of existing ones.

Our implementation of ADL is the result of the influence of ideas from several areas. First, work by Malcom [14], Meijer, Fokkinga, and Paterson [15, 9], and Cockett [3, 4] which describe how to capture patterns of recursion for a large class of algebraic types in a uniform way. Many of the theorems which our transformation algorithms are based upon can be found here, second, our previous work on type reflection [13, 16], and our work on program normalization[5, 6, 17].

8 Conclusion

The formalism outlined above combining normalization, *Zr* law calculation, and the *tupling lemma* with *beta* and *eta* contraction provides a theoretical basis for calculation based transformations. We conjecture that in addition to the techniques outlined above a handful of other techniques similar in generality can be found to increase the tools effectiveness. The ADL language and the transformation tool have been implemented, and are in use here at OGI.

References

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

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

[3] 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.

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

[5] 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.

[6] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In *Proceedings of the 11th International Conference on Automated*

¶our `red`

‖our `hom` with a simple algebra where all operators are replaced by their free constructors.

Deduction (CADE-11), Saratoga Springs, New York, pp 148–162. Springer-Verlag, June 1992.

[7] L. Fegaras, T. Sheard and T. Zhou. Improving Programs which Recurse over Multiple Inductive Structures. Oregon Graduate Institute, Technical report #94-005 A version of this paper is ftp-able from `cse.ogi.edu:/pub/pacsoft/papers/ImpProgRecMult.ps`. Submitted to PEPM'94.

[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] M.M. Fokkinga Calculate Categorically! Formal Aspects of Computing(1992) Vol 4, pp 673-692.

[10] M.M. Fokkinga, Tupling and Mutamorphisms, *The Squiggolist*, 1(4) 1989.

[11] A. Gill, J. Launchbury, and S. Peyton Jones A Short Cut to Deforestation In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993. pp 223-232

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

[13] J. Hook, R. Kieburtz, and T. Sheard. Generating Programs by Reflection. Oregon Graduate Institute Technical Report 92-015,

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

[15] 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.

[16] Tim Sheard. Type parametric programming. Technical Report 93-018, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[17] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[18] T. Sheard and L. Fegaras. Optimizing Algebraic Programs. OGI, Tech-report #94-004. The Extended version of this paper. Ftp-able from `cse.ogi.edu:/pub/pacsoft/papers/OptAlgProg.ps`. Submitted to PEPM'94.

[19] 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.

[20] 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.

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

A Correctness of the Normalization Algorithm

A reduction $\mathrm{red}[T]\{\$c_i := f_i\}$ is defined by the following rules:

$$\mathrm{red}[T]\{\$c_i := f_i\} \circ C_k \quad = \quad f_k \circ E\$c_k(\mathrm{red}[T]\{\$c_i := f_i\})$$

where for any constructor $C_k : \overline{\tau} \rightarrow T$ of T: $E\$c_i(f) = \mathcal{K}[T, \overline{\tau}](f)$. The combinator $\mathcal{K}$ is defined by the following inductive equations:

$$\begin{array}{lcll}
\mathcal{K}[T,T](f) & = & f & \\
\mathcal{K}[T,t_1 \times t_2](f) & = & \mathcal{K}[T,t_1](f) \times \mathcal{K}[T,t_2](f) & \\
\mathcal{K}[T,S(t_1,\ldots,t_q)](f) & = & \mathrm{map}^S(\mathcal{K}[T,t_1](f),\ldots,\mathcal{K}[T,t_q](f)) & \\
\mathcal{K}[T,t](f) & = & \mathrm{id} & \text{otherwise}
\end{array}$$

where map^S is a map over the type $S(\alpha_1, \ldots, \alpha_q)$, that is, it maps the parametric type $S(\alpha_1, \ldots, \alpha_q)$ into the type $S(\beta_1, \ldots, \beta_q)$. Thus $\mathrm{map}^S(f_1, \ldots, f_q)$ requires q functions $f_i : \alpha_i \rightarrow \beta_i$, one for each type variable.

Figure 1 displays the meaning function $\mathcal{M}[\![t]\!]\,\sigma$ that maps terms t of the language into values. The binding list σ is a partial function from term variables to values. We assume that all the free variables in the term t have a binding in σ. Figure 2 displays the normalization algorithm $\mathcal{N}[\![t]\!]$ that maps a term t into another term or possibly into nothing. That is, the type of $\mathcal{N}$ is $\mathrm{term} \rightarrow \mathrm{maybe(term)}$. Operation $t_1 \bowtie t_2$ returns either a pair of terms, if both t_1 and t_2 are terms, or nothing if any of them is nothing. Similarly, operation $f \diamond e$ applies f to e if both f and e are terms, otherwise it returns nothing:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathrm{just}(a) \bowtie \mathrm{just}(b) & = & \mathrm{just}(a,b) & \mathrm{just}(f) \diamond \mathrm{just}(a) & = & \mathrm{just}(f\ a) \\
\mathrm{nothing} \bowtie x & = & \mathrm{nothing} & \mathrm{nothing} \diamond x & = & \mathrm{nothing} \\
x \bowtie \mathrm{nothing} & = & \mathrm{nothing} & x \diamond \mathrm{nothing} & = & \mathrm{nothing}
\end{array}$$

We will show in Theorem 2 that the normalization algorithm preserves the meaning of a term. The only law that we will use to prove this theorem is the promotion theorem that describes a meaning preserving transformation. First we will prove an important property of the normalization algorithm. It says that a reduction behaves like a homomorphism under certain conditions. This condition is described in Figure 3 and states exactly the case in which the normalization algorithm does not construct a value 'nothing' at any level of normalization.

Theorem 1 (Homomorphic Property) *For any reduction $g : S \to \alpha$ and for any terms $e_i : \tau_i$ and $f : \overline{\tau} \to T$ such that $\mathcal{P}[\![\mathcal{K}[T,S](g)(f(\overline{e}))]\!]$ is true, there exists a term $\mathcal{X}$ that does not depend on g and e_i such that:*

$$\mathcal{N}[\![\mathcal{K}[T,S](g)(f(\overline{e}))]\!] \;\to\; \mathcal{N}[\![\mathcal{X}]\!] \diamond (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](g)\,\overline{e}]\!])$$

Proof: Let $g = \text{red}[T]\{\$c_i := g_i\}$ and $G = \mathcal{K}[T,S](g)$. We will use induction over the term f. There are three possible ways to form f: as a variable, as a reduction, and as a construction.
Induction base: If f has the form $\lambda\overline{z}.z_i$ then the theorem is true ($\mathcal{X}$ is identity).
Induction hypothesis: We will assume that the theorem is true for any subterm f' of f.
Induction step: We will prove the theorem for some term f:

- let $f = \lambda\overline{z}.\text{red}[R]\{\$c_i := \lambda\overline{w}.f_i(\overline{w},\overline{z})\}\; v(\overline{z})$. Then the normalization algorithm will use Rule (9) to construct a new reduction $\text{red}[R]\{\$c_i := \lambda\overline{w}.\phi_i\}$, since $\mathcal{P}[\![G(f(\overline{e}))]\!]$ is true.

$$\begin{aligned}\mathcal{N}[\![G(f(\overline{e}))]\!] &= \mathcal{N}[\![G(\text{red}[R]\{\$c_i := \lambda\overline{w}.f_i(\overline{w},\overline{e})\}\; v(\overline{e}))]\!]\\ &= \text{just}(\text{red}[R]\{\$c_i := \lambda\overline{w}.\phi_i\}) \diamond \mathcal{N}[\![v(\overline{e})]\!]\end{aligned}$$

  where

$$\begin{array}{rcll}\text{just}(\phi_i) & = & \mathcal{N}[\![G(f_i(E\$c_i(G^{-1})\,\overline{w},\overline{e})]\!] & \textit{from (9)}\\ & \to & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond (\mathcal{N}[\![E\$c_i(G)(E\$c_i(G^{-1})\,\overline{w})]\!] \bowtie (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](G)\,\overline{e}]\!])) & \textit{by hypothesis}\\ & \to & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond (\mathcal{N}[\![E\$c_i(G \circ G^{-1})\,\overline{w}]\!] \bowtie (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](G)\,\overline{e}]\!])) & \textit{by (10) and (11)}\\ & \to & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond ((\text{just}(E\$c_i) \diamond \mathcal{N}[\![G \circ G^{-1}]\!] \diamond \text{just}(\overline{w})) \bowtie (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](G)\,\overline{e}]\!])) & \textit{by (11)}\\ & \to & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond (\text{just}(\overline{w}) \bowtie (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](G)\,\overline{e}]\!])) & \textit{by (7)}\\ & = & \mathcal{N}[\![\lambda\overline{z}.\mathcal{X}_i(\overline{w},\overline{z})]\!] \diamond \mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](G)\,\overline{e}]\!] & \end{array}$$

  Now we need to prove that $\mathcal{N}[\![v(\overline{e})]\!]$ can be put into the form $\mathcal{N}[\![\mathcal{X}]\!] \diamond (\mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](g)\,\overline{e}]\!])$. This is possible only when $\mathcal{P}[\![G(f(\overline{e}))]\!]$ is true, that is, when $\mathcal{P}[\![v(\overline{e})]\!]$ is true.

- Let $f = \lambda\overline{z}.C_k(v_1(\overline{z}),\ldots,v_m(\overline{z}))$, where $C_k : s_1 \times \cdots \times s_m \to T$ is a constructor of T. We have two cases for type S:

  - $S = T$: then $\mathcal{K}[T,S](g) = g$ and

$$\begin{array}{rcll}\mathcal{N}[\![G(f(\overline{e}))]\!] & = & \mathcal{N}[\![\text{red}[T]\{\$c_i := g_i\}(C_k(v_1(\overline{e}),\ldots,v_m(\overline{e})))]\!] & \\ & \to & \mathcal{N}[\![g_i(E\$c_k(g)(v_1(\overline{e}),\ldots,v_m(\overline{e})))]\!] & \textit{(by (8))}\\ & = & \mathcal{N}[\![g_i(\mathcal{K}[T,s_1](g)(v_1(\overline{e})),\ldots,\mathcal{K}[T,s_n](g)(v_m(\overline{e})))]\!] & \\ & \to & \mathcal{N}[\![g_i]\!] \diamond (\mathcal{N}[\![\mathcal{K}[T,s_1](g)(v_1(\overline{e}))]\!] \bowtie \cdots \bowtie \mathcal{N}[\![\mathcal{K}[T,s_n](g)(v_m(\overline{e}))]\!]) & \\ & \to & \mathcal{N}[\![g_i]\!] \diamond ((\mathcal{N}[\![\mathcal{X}_1]\!] \diamond \mathcal{N}[\![\mathcal{K}[T,\tau_1](g)(\overline{e})]\!]) \bowtie \cdots \bowtie (\mathcal{N}[\![\mathcal{X}_m]\!] \diamond \mathcal{N}[\![\mathcal{K}[T,\tau_n](g)(\overline{e})]\!])) & \\ & & \textit{(by induction hypothesis)} & \\ & = & \mathcal{N}[\![g_i \circ (\mathcal{X}_1 \times \cdots \mathcal{X}_n)]\!] \diamond \mathcal{N}[\![\mathcal{K}[T,\overline{\tau}](g)(\overline{e})]\!] & \end{array}$$

– $S = R(r_1, \ldots, r_q)$: then $\mathcal{K}[T, S](g) = \mathrm{map}^R(\overline{\phi})$, where $\phi_i = \mathcal{K}[T, r_i](g)$. Then

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{N}[\![\mathrm{map}^R(\overline{\phi})(C_k(v_1(\overline{e}), \ldots, v_m(\overline{e})))]\!]} \\ = & \mathcal{N}[\![C_k(\mathcal{K}[T, r_1](\mathrm{map}^R(\overline{\phi}))(v_1(\overline{e})), \ldots, \mathcal{K}[T, r_m](\mathrm{map}^R(\overline{\phi}))(v_m(\overline{e})))]\!] \\ = & \mathcal{N}[\![C_k(\mathcal{K}[T, s_1](g)(v_1(\overline{e})), \ldots, \mathcal{K}[T, s_m](g)(v_m(\overline{e})))]\!] \\ \to & \mathcal{N}[\![C_k]\!] \diamond (\mathcal{N}[\![\mathcal{K}[T, s_1](g)(v_1(\overline{e}))]\!] \bowtie \cdots \bowtie \mathcal{N}[\![\mathcal{K}[T, s_n](g)(v_m(\overline{e}))]\!]) \\ \to & \mathcal{N}[\![C_k]\!] \diamond ((\mathcal{N}[\![\mathcal{X}_1]\!] \diamond \mathcal{N}[\![\mathcal{K}[T, \tau_1](g)(\overline{e})]\!]) \bowtie \cdots \bowtie (\mathcal{N}[\![\mathcal{X}_m]\!] \diamond \mathcal{N}[\![\mathcal{K}[T, \tau_n](g)(\overline{e})]\!])) \\ & \textit{(by induction hypothesis)} \\ = & \mathcal{N}[\![C_k \circ (\mathcal{X}_1 \times \cdots \mathcal{X}_n)]\!] \diamond \mathcal{N}[\![\mathcal{K}[T, \overline{\tau}](g)(\overline{e})]\!] \end{array}$$

□

Theorem 2 (Correctness of the Normalization Algorithm)

$$\forall t \forall \sigma \exists s : \ \mathcal{N}[\![t]\!] \to \mathrm{just}(s) \ \Rightarrow \ \mathcal{M}[\![t]\!]\,\sigma = \mathcal{M}[\![s]\!]\,\sigma$$

Proof: It is easy to prove that all rules except Rule (9) are meaning preserving. For Rule (9) we need to prove that

$$\mathcal{M}[\![g(\mathrm{red}[T]\{\$c_i := f_i\}\, e)]\!]\,\sigma \ = \ (\mathcal{M}[\![\mathrm{red}[T]\{\$c_i := \phi_i\}]\!]\,\sigma)(\mathcal{M}[\![e]\!]\,\sigma)$$

where for any i:

$$\mathrm{just}(\phi_i) \ = \ \mathcal{N}[\![\lambda \overline{x}.g(f_i(E\$c_i(g^{-1})\,\overline{x}))]\!]$$

That is, according to the promotion theorem we need to prove that for any σ:

$$\mathcal{M}[\![\mathcal{N}[\![g(f_i(\overline{x}))]\!]]\!]\,\sigma \ = \ \mathcal{M}[\![\mathcal{N}[\![\phi_i(E\$c_i(g)\,\overline{x})]\!]]\!]\,\sigma$$

We use Theorem 1 to normalize the left part of the premise of the promotion theorem (since $\mathcal{P}[\![g(f_i(\overline{x}))]\!]$ is true):

$$\begin{array}{lll} \mathcal{N}[\![g(f_i(\overline{x}))]\!] & \to & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond \mathcal{N}[\![\mathcal{K}[T, \overline{\tau}](g)\,\overline{x}]\!] \quad \textit{from Theorem 1} \\ & = & \mathcal{N}[\![\mathcal{X}_i]\!] \diamond \mathcal{N}[\![E\$c_i(g)\,\overline{x}]\!] \end{array}$$

Therefore, if $\mathrm{just}(\phi_i) = \mathcal{N}[\![\mathcal{X}_i]\!]$ then

$$\mathcal{M}[\![\mathcal{N}[\![g(f_i(\overline{x}))]\!]]\!]\,\sigma \ = \ \mathcal{M}[\![\mathcal{N}[\![\mathcal{X}_i]\!] \diamond \mathcal{N}[\![E\$c_i(g)\,\overline{x}]\!]]\!]\,\sigma \ = \ \mathcal{M}[\![\mathcal{N}[\![\phi_i(E\$c_i(g)\,\overline{x})]\!]]\!]\,\sigma$$

which makes the promotion theorem valid. □

$$
\begin{array}{lcl}
\mathcal{M}[\![v]\!]\,\sigma & = & \sigma[v] \\
\mathcal{M}[\![C_i]\!]\,\sigma & = & C_i \\
\mathcal{M}[\![\lambda x.e]\!]\,\sigma & = & \lambda z.\mathcal{M}[\![e]\!]\,\sigma[z/x] \\
\mathcal{M}[\![(e_1,e_2)]\!]\,\sigma & = & (\mathcal{M}[\![e_1]\!]\,\sigma, \mathcal{M}[\![e_2]\!]\,\sigma) \\
\mathcal{M}[\![\mathrm{red}[T]\{\$c_i := f_i\}\,e]\!]\,\sigma & = & \left\{\begin{array}{l}
\textbf{case}\ \mathcal{M}[\![e]\!]\,\sigma\ \textbf{of} \\
\quad\ \ \ldots \\
\mid\ \ C_k(\overline{v})\ \Rightarrow\ \mathcal{M}[\![f_k(E\$c_k(\mathrm{red}[T]\{\$c_i := f_i\})\,\overline{z})]\!]\,\sigma[\overline{v}/\overline{z}] \\
\quad\ \ \ldots
\end{array}\right. \\
\mathcal{M}[\![f\,e]\!]\,\sigma & = & (\mathcal{M}[\![f]\!]\,\sigma)\,(\mathcal{M}[\![e]\!]\,\sigma)
\end{array}
$$

Figure 1: The Evaluation Algorithm

$$
\begin{array}{rcll}
\mathcal{N}[\![g^{-1}]\!] & \rightarrow & \mathrm{nothing} & (1) \\
\mathcal{N}[\![v]\!] & \rightarrow & \mathrm{just}(v) & (2) \\
\mathcal{N}[\![C_i]\!] & \rightarrow & \mathrm{just}(C_i) & (3) \\
\mathcal{N}[\![(e_1,e_2)]\!] & \rightarrow & \mathcal{N}[\![e_1]\!] \bowtie \mathcal{N}[\![e_2]\!] & (4) \\
\mathcal{N}[\![\lambda x.e]\!] & \rightarrow & \mathrm{just}(\lambda z.\lambda x.z) \diamond \mathcal{N}[\![e]\!] & (5) \\
\mathcal{N}[\![\mathrm{red}[T]\{\$c_i := f_i\}]\!] & \rightarrow & \mathrm{just}(\lambda\overline{h}.\mathrm{red}[T]\{\$c_i := h_i\}) \diamond (\mathcal{N}[\![f_1]\!] \bowtie \cdots \bowtie \mathcal{N}[\![f_n]\!]) & (6) \\
\mathcal{N}[\![g(h^{-1}(x))]\!] & \rightarrow & \textbf{if}\ g = h\ \textbf{then}\ \mathrm{just}(x)\ \textbf{else}\ \mathrm{nothing} & (7) \\
\mathcal{N}[\![\mathrm{red}[T]\{\$c_i := f_i\}\,(C_i\,e)]\!] & \rightarrow & \mathcal{N}[\![f_i(E\$c_i(\mathrm{red}[T]\{\$c_i := f_i\})\,e)]\!] & (8) \\
\mathcal{N}[\![g(\mathrm{red}[T]\{\$c_i := f_i\}\,e)]\!] & \rightarrow & \left\{\begin{array}{l}
\textbf{case}\ \mathcal{N}[\![e]\!] \bowtie \mathcal{N}[\![h_1]\!] \bowtie \cdots \bowtie \mathcal{N}[\![h_n]\!] \\
\qquad \textbf{where}\ h_i = \lambda\overline{x}.g(f_i(E\$c_i(g^{-1})\,\overline{x}))\ \textbf{of} \\
\quad\ \mathrm{just}((x,\phi_1),\ldots,\phi_n)\ \Rightarrow\ \mathrm{just}(\mathrm{red}[T]\{\$c_i := \phi_i\}\,x) \\
\mid\ \ \mathrm{nothing}\ \Rightarrow\ \mathcal{N}[\![g]\!] \diamond \mathcal{N}[\![\mathrm{red}[T]\{\$c_i := f_i\}\,e]\!]
\end{array}\right. & (9) \\
\mathcal{N}[\![(\lambda v.e)\,t]\!] & \rightarrow & \mathcal{N}[\![\mathrm{beta}(v,e,t)]\!] & (10) \\
\mathcal{N}[\![f\,e]\!] & \rightarrow & \mathcal{N}[\![f]\!] \diamond \mathcal{N}[\![e]\!] & (11)
\end{array}
$$

Figure 2: The Normalization Algorithm

$$
\begin{array}{lcl}
\mathcal{P}[\![g^{-1}]\!] & = & \text{false} \\
\mathcal{P}[\![v]\!] & = & \text{true} \\
\mathcal{P}[\![C_i]\!] & = & \text{true} \\
\mathcal{P}[\![(e_1, e_2)]\!] & = & \mathcal{P}[\![e_1]\!] \wedge \mathcal{P}[\![e_2]\!] \\
\mathcal{P}[\![\lambda x.e]\!] & = & \mathcal{P}[\![e]\!] \\
\mathcal{P}[\![\text{red}[T]\{\$c_i := f_i\}]\!] & = & \mathcal{P}[\![f_i]\!] \\
\mathcal{P}[\![g(h^{-1}(x))]\!] & = & (g = h) \\
\mathcal{P}[\![\text{red}[T]\{\$c_i := f_i\}\,(C_i\,e)]\!] & = & \mathcal{P}[\![f_i(E\$c_i(\text{red}[T]\{\$c_i := f_i\})\,e)]\!] \\
\mathcal{P}[\![g(\text{red}[T]\{\$c_i := f_i\}\,e)]\!] & = & \mathcal{P}[\![e]\!] \wedge \mathcal{P}[\![h_1]\!] \wedge \cdots \wedge \mathcal{P}[\![h_n]\!] \\
 & & \textbf{where } h_i = \lambda\overline{x}.g(f_i(E\$c_i(g^{-1})\,\overline{x})) \\
\mathcal{P}[\![(\lambda v.e)\,t]\!] & = & \mathcal{P}[\![\text{beta}(v, e, t)]\!] \\
\mathcal{P}[\![f\,e]\!] & = & \mathcal{P}[\![f]\!] \wedge \mathcal{P}[\![e]\!]
\end{array}
$$

Figure 3: The Promotion Condition