Fusion for Free!

OGI, Tech-report #96-001

Leonidas Fegaras
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@cse.ogi.edu

January 9, 1996

Abstract

Program fusion techniques have long been proposed as an effective means of improving program performance and of eliminating unnecessary intermediate data structures. This paper proposes a new approach on program fusion that is based entirely on the type signatures of programs. First, for each function, a recursive skeleton is extracted that captures its pattern of recursion. Then, the parametricity theorem of this skeleton is derived, which provides a rule for fusing this function with any function. This method generalizes other approaches that use fixed parametricity theorems to fuse programs.

1 Introduction

There is much work recently on using higher-order operators, such as *fold* [9] and *build* [6, 5], to automate program fusion [2] and deforestation [11]. Even though these methods do a good job on fusing programs, they are only effective if programs are expressed in terms of these operators. This limits their applicability to conventional functional languages. To ameliorate this problem, some researchers proposed methods to translate regular functional programs into folds [6]. These methods had a moderate success so far, and only for simple functions.

The main reason for using these higher-order operators is that they satisfy some powerful theorems, which facilitate program optimization. But there is nothing special about these theorems. They are parametricity theorems [10] that are derived exclusively from the types of these operators. Any function satisfies a parametricity theorem. The difference is that most functions are not *sufficiently polymorphic* and, thus, their parametricity theorems are usually trivial.

This paper proposes a new approach on fusing programs. Instead of trying to express a function in terms of a particular higher-order operator, such as fold, we generate an individualized higher-order operator for this function. This operator, called the *recursive skeleton* of this function,

$$\begin{tikzcd}
\alpha_2 & \mathrm{list}(\alpha_2) & \mathrm{list}(\mathrm{list}(\alpha_2)) & \mathrm{list}(\mathrm{list}(\alpha_2)) \arrow[r, "\mathrm{flat}"] & \mathrm{list}(\alpha_2) \\
\alpha_1 \arrow[u, "f"'] & \mathrm{list}(\alpha_1) \arrow[u, "\mathrm{map}(f)"'] & \mathrm{list}(\mathrm{list}(\alpha_1)) \arrow[u, "\mathrm{map}(\mathrm{map}(f))"'] & \mathrm{list}(\mathrm{list}(\alpha_1)) \arrow[u, "\mathrm{map}(\mathrm{map}(f))"'] \arrow[r, "\mathrm{flat}"'] & \mathrm{list}(\alpha_1) \arrow[u, "\mathrm{map}(f)"']
\end{tikzcd}$$

Functor: list *Functor:* list $\circ$ list *Natural transformation:* list $\circ$ list $\dot{\to}$ list

Figure 1: The Parametricity Theorem for flat : $\forall\alpha.\,\mathrm{list}(\mathrm{list}(\alpha))\to\mathrm{list}(\alpha)$

captures its pattern of recursion. It is specific to this function only and may not be suitable for any other function. This operator is polymorphic enough to satisfy a useful parametricity theorem, which is very similar to the one for fold. In fact, if a function resembles a fold, then its recursive skeleton is exactly the fold operator. For each such recursive skeleton, we generate the parametricity theorem. Program fusion is achieved by using these theorems alone. In a way, our method generalizes all other methods that use the parametricity theorems of a fixed set of higher-order operators. It is also directly applicable to conventional functional programs. The drawback though is that our method uses many different theorems for program fusion, instead of just a fixed number. But, as we will see in this paper, using these theorems is actually no harder than using the parametricity theorems for folds.

We believe that our approach may well turned out to have practical uses for optimizing real functional languages. We also believe that it can be useful for proving equational theorems about functions.

2 Background: The Parametricity Theorem

Any function f of type τ satisfies a parametricity theorem (also called *theorem for free* [10]), which is derived directly from the type τ. For first-order functions, this theorem states that any polymorphic function is a natural transformation. For example, Figure 1 gives the parametricity theorem for any function *flat* of type:

$$\forall\alpha.\,\mathrm{list}(\mathrm{list}(\alpha))\to\mathrm{list}(\alpha)$$

The parametric type list(α) is a functor that maps any type α_i into the type list(α_i) and any function f of type $\alpha_1\to\alpha_2$ into the function map(f) of type list$(\alpha_2)\to$list(α_2). In general, any parametric type $T(\alpha)$ is a functor that maps a function f into $\mathrm{map}^T(f)$, where $\mathrm{map}^T : (\alpha\to\beta)\to T(\alpha)\to T(\beta)$ is the map function for type T. Since the composition of functors is also a functor, the type list(list(α)) is a functor that maps f into map(map(f)). The parametricity theorem for *flat* is the commuting diagram in Figure 1. It can be expressed as follows:

$$\forall f:\ \text{flat} \circ \text{map}(\text{map}(f)) = \text{map}(f) \circ \text{flat}$$

This commuting diagram represents a natural transformation between the functors list $\circ$ list and list. It indicates that applying f to every element of a nested list and then flattening the resulting nested list is equivalent to flattening the list and then applying f to the flat list. This theorem is always true regardless of the actual definition of *flat* because, if f were changing in some way the elements of the nested list, then the type of *flat* would not be the polymorphic type given above. (We assume that all functions are strict here, as they would be if they were defined in a non-lazy language.)

The proof of this theorem comes directly from [10]):

Theorem 1 (Parametricity Theorem) *Any expression $e:\tau$ satisfies $\mathcal{P}[\![\tau]\!](e,e)\,\rho$, where:*

$$\begin{array}{lll}
\mathcal{P}[\![\text{basic}]\!](r,s)\,\rho & \rightarrow & r = s \\
\mathcal{P}[\![\alpha]\!](r,s)\,\rho & \rightarrow & r = \rho(\alpha)(s) \\
\mathcal{P}[\![\forall\alpha.\,\tau]\!](r,s)\,\rho & \rightarrow & \forall f_\alpha:\ \mathcal{P}[\![\tau]\!](r,s)\,\rho[f_\alpha/\alpha] \\
\mathcal{P}[\![\tau_1\times\tau_2]\!](r,s)\,\rho & \rightarrow & \mathcal{P}[\![\tau_1]\!](\pi_1(r),\pi_1(s))\,\rho \wedge \mathcal{P}[\![\tau_2]\!](\pi_2(r),\pi_2(s))\,\rho \\
\mathcal{P}[\![\tau_1\rightarrow\tau_2]\!](r,s)\,\rho & \rightarrow & \forall x,y:\ \mathcal{P}[\![\tau_1]\!](x,y)\,\rho \Rightarrow \mathcal{P}[\![\tau_2]\!](r(x),s(y))\,\rho \\
\mathcal{P}[\![T(\tau)]\!](r,s)\,\rho & \rightarrow & \forall f:\ (\forall x:\mathcal{P}[\![\tau]\!](f(x),x)\,\rho) \Rightarrow r = \text{map}^T(f)\,s
\end{array}$$

That is, for each type variable α, we associate a function f_α of type $\alpha_1\rightarrow\alpha_2$, where α_1 and α_2 are instances of α.

To illustrate Theorem 1, we derive the parametricity theorem for the list fold:

$$\text{fold}:\ \forall\alpha,\beta.\ (\alpha\rightarrow\beta\rightarrow\beta)\rightarrow\beta\rightarrow\text{list}(\alpha)\rightarrow\beta$$

The construction is accomplished in four simple steps:

$$\begin{array}{lll}
\mathcal{P}[\![\forall\delta,\varepsilon.\ \delta\rightarrow\varepsilon]\!](r,s)\,\rho & \rightarrow & \forall f_\delta,f_\varepsilon,x,y:\ \mathcal{P}[\![\delta]\!](x,y)\,\rho[f_\delta/\delta,f_\varepsilon/\varepsilon] \\
& & \qquad\qquad \Rightarrow \mathcal{P}[\![\varepsilon]\!](r(x),s(y))\,\rho[f_\delta/\delta,f_\varepsilon/\varepsilon] \\
& \rightarrow & \forall f_\delta,f_\varepsilon,x,y:\ x = f_\delta(y) \Rightarrow r(x) = f_\varepsilon(s(y)) \\
& \text{or} & \forall f_\delta,f_\varepsilon:\ r\circ f_\delta = f_\varepsilon\circ s \\
\mathcal{P}[\![\forall\delta,\varepsilon,\eta.\ \delta\rightarrow\varepsilon\rightarrow\eta]\!](r,s)\,\rho & \rightarrow & \forall f_\delta,f_\varepsilon,f_\eta,x,y:\ \mathcal{P}[\![\delta]\!](x,y)\,\rho[f_\delta/\delta,f_\varepsilon/\varepsilon,f_\eta/\eta] \\
& & \qquad\qquad \Rightarrow \mathcal{P}[\![\varepsilon\rightarrow\eta]\!](r(x),s(y))\,\rho[f_\delta/\delta,f_\varepsilon/\varepsilon,f_\eta/\eta] \\
& \rightarrow & \forall f_\delta,f_\varepsilon,f_\eta,x,y:\ x = f_\delta(y) \Rightarrow r(x)\circ f_\varepsilon = f_\eta\circ s(y) \\
& \text{or} & \forall f_\delta,f_\varepsilon,f_\eta,y,z:\ r\ (f_\delta\ y)\ (f_\varepsilon\ z) = f_\eta(s\ y\ z) \\
\mathcal{P}[\![\forall\alpha.\ \text{list}(\alpha)]\!](r,s)\,\rho & \rightarrow & \forall f_\alpha,g:\ (\forall x: g(x) = f_\alpha(x)) \Rightarrow r = \text{map}(g)\,s \\
& \text{or} & \forall f_\alpha:\ r = \text{map}(f_\alpha)\,s
\end{array}$$

$$\begin{aligned}&\mathcal{P}[\![\forall\alpha,\beta.\,(\alpha\to\beta\to\beta)\to\beta\to\text{list}(\alpha)\to\beta]\!](r,s)\,\rho\\ &\quad\to\ \mathcal{P}[\![\alpha\to\beta\to\beta]\!](\otimes,\oplus)\,\rho[f_\alpha/\alpha,\,f_\beta/\beta]\ \Rightarrow\ \mathcal{P}[\![\beta\to\text{list}(\alpha)\to\beta]\!](r\,\otimes,\,s\,\oplus)\,\rho[f_\alpha/\alpha,\,f_\beta/\beta]\\ &\quad\to\ \forall f_\alpha,f_\beta,x,y:\ (f_\alpha\,x)\otimes(f_\beta\,y)=f_\beta(x\oplus y)\ \Rightarrow\ r\,\otimes\,(f_\beta\,x)\,(\text{map}(f_\alpha)\,y)=f_\beta(s\,\oplus\,x\,y)\end{aligned}$$

That is, fold satisfies the following theorem:

$$\forall f_\alpha,f_\beta,x,y:\ \ (f_\alpha\,x)\otimes(f_\beta\,y)=f_\beta(x\oplus y)\quad\Rightarrow\quad\text{fold}(\otimes)\,(f_\beta\,x)\,(\text{map}(f_\alpha)\,y)=f_\beta(\text{fold}(\oplus)\,x\,y)$$

If we set $f_\alpha=\text{id}$, where $\text{id}=\lambda x.x$, we get:

$$\forall f_\beta,x,y:\qquad x\otimes(f_\beta\,y)=f_\beta(x\oplus y)\ \ \Rightarrow\ \ \text{fold}(\otimes)\,(f_\beta\,x)\,y=f_\beta(\text{fold}(\oplus)\,x\,y)\tag{1}$$

which is the fusion law for list fold [9]. Binding the function f_α, which corresponds to the type variable α, to id is something that we will do often in this paper. In general, if we have a type t that depends on some type variables α, β, etc., i.e., t has the form $t(\alpha,\beta,\ldots)$, then finding the parametricity theorem for $t(\alpha,\beta,\ldots)$ and then setting $f_\alpha=\text{id}$ is the same as finding the parametricity theorem for the type $t((),\beta,\ldots)$, where $()$ is the unit type.

Another example is the monad extension operator:

$$\text{ext}^T\ :\ \forall\alpha,\beta.\,(\alpha\to T(\beta))\to T(\alpha)\to T(\beta)$$

that satisfies for $\beta=()$:

$$\forall f_\alpha,g:\ \ \text{ext}^T(g)\circ\text{map}^T(f_\alpha)=\text{ext}^T(g\circ f_\alpha)$$

The Y combinator for functions is defined as $Y(f)\,x=f(Y(f))\,x$ and has type:

$$\forall\alpha,\beta.\,((\alpha\to\beta)\to\alpha\to\beta)\to\alpha\to\beta$$

The parametricity theorem for this type with $\alpha=()$ is:

$$\forall f_\beta,f,f',g:\qquad f_\beta\circ(f\,g)=f'(f_\beta\circ g)\ \ \Rightarrow\ \ f_\beta\circ(Y\,f)=Y\,f'\tag{2}$$

which is actually the unfold-simplify-fold law [1].

3 Expressing the Parametricity Theorem using Bifunctors

A bifunctor is a generalization of a functor. In contrast to regular functors, bifunctors can capture types with contravariant type variables [3].

Definition 1 (Bifunctor) *Let* $\forall\alpha_1,\ldots,a_n:\ \tau$ *be a type and let* $\overline{f}=f_1,\ldots,f_n$ *and* $\overline{g}=g_1,\ldots,g_n$*. The* bifunctor $\mathcal{F}[\![\tau]\!](\overline{f},\overline{g})$ *is defined as follows:*

$$\begin{aligned}\mathcal{F}[\![\text{basic}]\!](\overline{f},\overline{g})&\quad\to\quad\text{id}\\ \mathcal{F}[\![\alpha_i]\!](\overline{f},\overline{g})&\quad\to\quad f_i\\ \mathcal{F}[\![\tau_1\times\tau_2]\!](\overline{f},\overline{g})&\quad\to\quad\mathcal{F}[\![\tau_1]\!](\overline{f},\overline{g})\times\mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\\ \mathcal{F}[\![\tau_1\to\tau_2]\!](\overline{f},\overline{g})&\quad\to\quad\lambda h.\,\mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f})\\ \mathcal{F}[\![T(\tau)]\!](\overline{f},\overline{g})&\quad\to\quad\text{map}^T(\mathcal{F}[\![\tau]\!](\overline{f},\overline{g}))\end{aligned}$$

where the product of functions is defined by $(f \times g)(x,y) = (f\,x, g\,y)$. For example,

$$\mathcal{F}[\![(\alpha \rightarrow \alpha) \rightarrow \alpha]\!](f,g) \;=\; \lambda h.\, f \circ h \circ (\lambda k.\, g \circ k \circ f)$$

It is easy to prove the following theorem:

Theorem 2 *A bifunctor $\mathcal{F}[\![\tau]\!](\overline{f},\overline{g})$ is a functor that is covariant over f_i and contravariant over g_i. That is,*

$$\mathcal{F}[\![\tau]\!](\overline{\mathrm{id}},\overline{\mathrm{id}}) \;=\; \mathrm{id} \qquad (3)$$

$$\mathcal{F}[\![\tau]\!](\overline{f},\overline{g}) \circ \mathcal{F}[\![\tau]\!](\overline{f'},\overline{g'}) \;=\; \mathcal{F}[\![\tau]\!](\overline{f \circ f'},\overline{g' \circ g}) \qquad (4)$$

The following theorem expresses the parametricity theorem of a type in terms of the type's bifunctor (the proof is given in the appendix):

Theorem 3 *For any type $\forall \alpha_1, \ldots, a_n : \tau$ we have:*

$$\forall f_i, g_i, x, y : \;\; f_i \circ g_i = \mathrm{id} \;\;\Rightarrow\;\; (\mathcal{P}[\![\tau]\!](x,y)[\overline{f_i/\alpha_i}] \;\;\Leftrightarrow\;\; x = \mathcal{F}[\![\tau]\!](\overline{f},\overline{g})\; y)$$

For example, the bifunctor for $\tau = (\alpha \rightarrow \beta) \rightarrow \gamma$ is:

$$\mathcal{F}[\![\tau]\!](f_\alpha, f_\beta, f_\gamma, g_\alpha, g_\beta, g_\gamma) \;=\; \lambda h.\, f_\gamma \circ h \circ (\lambda k.\, g_\beta \circ k \circ f_\alpha)$$

and the parametricity theorem is:

$$\begin{array}{lll}
 & \mathcal{P}[\![\tau]\!](x,y)[f_\alpha/\alpha, f_\beta/\beta, f_\gamma/\gamma] & \\
\Leftrightarrow & \forall m, n :\; m \circ f_\alpha = f_\beta \circ n \Rightarrow x(m) = f_\gamma(y(n)) & \\
\Leftrightarrow & \forall m, n :\; m \circ f_\alpha \circ g_\alpha = f_\beta \circ n \circ g_\alpha \Rightarrow x(m) = f_\gamma(y(n)) & (e1) \\
\Leftrightarrow & \forall m, n :\; m = f_\beta \circ n \circ g_\alpha \Rightarrow x(m) = f_\gamma(y(n)) & \\
\Leftrightarrow & \forall n :\; x(f_\beta \circ n \circ g_\alpha) = f_\gamma(y(n)) & \\
\Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ n \circ g_\alpha) = f_\gamma \circ y & \\
\Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ n \circ g_\alpha) \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) & (e2) \\
\Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ g_\beta \circ n \circ f_\alpha \circ g_\alpha) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) & \\
\Leftrightarrow & x \circ (\lambda n.\, n) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) & \\
\Leftrightarrow & x = \mathcal{F}[\![\tau]\!](f_\alpha, f_\beta, f_\gamma, g_\alpha, g_\beta, g_\gamma)\; y &
\end{array}$$

The equivalences in $(e1)$ and $(e2)$ are based on the fact that $\mathrm{range}(h) = \mathrm{domain}(f) \;\Rightarrow\; (f = g \Leftrightarrow f \circ h = g \circ h)$.

We annotate type variables by a sign $s \in \{+,-\}$ as follows:

$$\begin{array}{lll}
\mathrm{basic}^s & = & \mathrm{basic} \\
\alpha^s & = & \alpha^s \\
(\tau_1 \times \tau_2)^s & = & \tau_1^s \times \tau_2^s \\
(\tau_1 \rightarrow \tau_2)^s & = & \tau_1^{\neg s} \rightarrow \tau_2^s \\
(T(\tau))^s & = & T(\tau^s)
\end{array}$$

where $\neg(+) = -$ and $\neg(-) = +$. For example,

$$((\alpha \to \beta) \to \gamma \to \delta)^+ \;=\; (\alpha^+ \to \beta^-) \to \gamma^- \to \delta^+$$

A type variable α is positive (resp., negative) in a type τ if all occurrences of α in τ^+ are α^+ (resp., α^-). For example, α and δ are positive in the type $(\alpha \to \beta) \to \gamma \to \delta$, while β and γ are negative.

It is easy to prove that if $f_i : \alpha_i^- \to \alpha_i^+$ and $g_i : \alpha_i^+ \to \alpha_i^-$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) : \tau^- \to \tau^+$.

Theorem 4 *Let* $\forall \alpha_1, \ldots, a_n : \tau$ *be a type whose type variables are positive in* τ*, then:*

$$\forall f_i, x, y : \;\; \mathcal{P}[\![\tau]\!](x, y)[\overline{f_i/\alpha_i}] \;\;\Leftrightarrow\;\; x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{\mathrm{id}})\, y$$

Proof: Since all type variables α_i are positive in τ, none of the g_is in Th. 3 is used. Therefore, each g_i can be replaced by an arbitrary function, including the identity function itself. $\square$

For example, the bifunctor for $\tau = \mathrm{int} \to \alpha \times (\mathrm{int} \to \mathrm{list}(\alpha))$ is:

$$\mathcal{F}[\![\tau]\!](f, g) = \lambda h.\, (f \times (\lambda k.\, \mathrm{map}(f) \circ k)) \circ h$$

We have $\mathcal{P}[\![\mathrm{int} \to \mathrm{list}(\alpha)]\!](g', g)[f/\alpha] \Leftrightarrow g' = \mathrm{map}(f) \circ g$ and

$$\begin{aligned} \mathcal{P}[\![\tau]\!](h', h)[f/\alpha] \;\;&\Leftrightarrow\;\; x = y \Rightarrow (\pi_1(h'(x)) = f(\pi_1(h(y)))) \wedge (\pi_2(h'(x)) = \mathrm{map}(f) \circ (\pi_2(h(y)))) \\ &\Leftrightarrow\;\; h'(x) = (f \times (\lambda k.\, \mathrm{map}(f) \circ k))\, (h(x)) \end{aligned}$$

which is equivalent to $h' = \mathcal{F}[\![\tau]\!](f, g)\, h$.

4 Using the Parametricity Theorem for Program Fusion

Consider the following non-polymorphic function of type $\mathrm{list}(\mathrm{int}) \to \mathrm{list}(\mathrm{int})$:

```
inc [ ]    = [ ]
inc (a:x)  = (1+a):(inc x)
```

The parametricity theorem of a non-polymorphic type is always a tautology. Luckily, inc happens to be a fold, since a fold has a similar pattern of recursion:

```
fold f b [ ]    = b
fold f b (a:x)  = f a (fold f b x)
```

In particular, inc x = fold(λa. λr. (1+a):r) [ ] x. This is quite useful because we know that fold satisfies a powerful parametricity theorem (Eq. 1). In fact, we have shown elsewhere [9] that there is an automated method for fusing a function composed with a fold. Suppose, for example that we want to fuse len(inc x), where len computes the length of a list, so that the intermediate list produced by inc and consumed by len is eliminated. Function len is defined as follows:

```
len [ ]    = 0
len(a:x)   = 1+(len x)
```

Thus, len(inc x) can be calculated from Eq. 1, where f_β = len, a $\oplus$ r=(1+a):r, x=[ ], and y=x. From the conclusion of Eq. 1 we have:

$$\begin{aligned} \mathsf{len(inc\ x)} &= \mathsf{len(fold(}\lambda\mathsf{a.\ }\lambda\mathsf{r.\ (1{+}a){:}r)\ [\ ]\ x)} \\ &= \mathsf{fold(}\otimes\mathsf{)\ (len\ [\ ])\ x} \\ &= \mathsf{fold(}\otimes\mathsf{)\ 0\ x} \end{aligned}$$

where $\otimes$ can be calculated from the premise of Eq. 1:

$$\begin{aligned} \mathsf{a} \otimes \mathsf{(len\ r)} &= \mathsf{len((1{+}a){:}r)} \\ &= \mathsf{1{+}(len\ r)} \end{aligned}$$

If we substitute (len r) for s, we get a $\otimes$ s=1+s and, finally,

$$\mathsf{len(inc\ x)} = \mathsf{fold(}\lambda\mathsf{a.\ }\lambda\mathsf{s.\ 1{+}s)\ 0\ x}$$

The resulting fold does not create the intermediate list of the original program.

Unfortunately, not all functions can be expressed as folds. Even though there are methods for translating a number of recursive functions into folds [6], these methods usually fail for complex functions. One solution to this problem is to use a list traversal scheme that is more flexible and maybe more expressive than fold. Some researchers have suggested hylomorphisms as a possible solution [7]. It remains an open issue of how easy it is to translate functions into hylomorphisms.

In this paper we propose an alternative solution to the above problem: instead of trying to make some recursive function fit the recursion pattern of a particular fixed traversal scheme, such as fold, we generate a traversal scheme that is individually tailored to this particular function. This scheme may not be useful for any other function. This traversal scheme is 'polymorphic enough' to satisfy a useful parametricity theorem. We can make a function more polymorphic (i.e., with more type variables) by abstracting pieces of its code into some extra function arguments. But when a function becomes 'polymorphic enough'? To answer this question we consider the parametricity theorem for fold. This theorem is useful because it has the conclusion:

$$f_\beta(\mathrm{fold}(\oplus)\ x\ y) \ = \ \mathrm{fold}(\otimes)\ (f_\beta\ x)\ y$$

The left part is the composition of any function f_β (which corresponds to the type variable β) with a fold. The right part is another fold whose arguments can be calculated from the arguments of the first fold by using the equalities in the premise of the theorem. That way we can fuse any function f_β with a fold yielding another fold. Given how the parametricity theorem should look like to be useful for fusion, we can easily guess how the type of a traversal scheme should look like to generate such a theorem: if the type of a traversal scheme f has the form $t_1 \to t_2 \to \cdots \to t_{n-1} \to t_n$, then t_n should be a type variable, say β. To see why, we derive the parametricity theorem for f from Theorem 1 (universal quantifications are omitted):

$$\begin{aligned} &\mathcal{P}[\![t_1 \to t_2 \to \cdots \to t_n]\!](f, f)\,\rho \\ &= \mathcal{P}[\![t_1]\!](x_1, y_1)\,\rho \Rightarrow \mathcal{P}[\![t_2 \to \cdots \to t_n]\!](f\,x_1, f\,y_1)\,\rho \\ &= \mathcal{P}[\![t_1]\!](x_1, y_1)\,\rho \Rightarrow (\mathcal{P}[\![t_2]\!](x_2, y_2)\,\rho \Rightarrow \mathcal{P}[\![t_3 \to \cdots \to t_n]\!](f\,x_1\,x_2, f\,y_1\,y_2)\,\rho) \\ &= \mathcal{P}[\![t_1]\!](x_1, y_1)\,\rho \Rightarrow (\mathcal{P}[\![t_2]\!](x_2, y_2)\,\rho \Rightarrow \cdots (\mathcal{P}[\![t_{n-1}]\!](x_{n-1}, y_{n-1})\,\rho \\ &\qquad \Rightarrow \mathcal{P}[\![t_n]\!](f\,x_1\,x_2 \ldots x_n, f\,y_1\,y_2 \ldots y_n)\,\rho)) \end{aligned}$$

If $t_n = \beta$, then the parametricity theorem for f becomes:

$$\begin{aligned}\mathcal{P}[\![t_1]\!](x_1, y_1)\,\rho \Rightarrow (\mathcal{P}[\![t_2]\!](x_2, y_2)\,\rho \Rightarrow \cdots (\mathcal{P}[\![t_{n-1}]\!](x_{n-1}, y_{n-1})\,\rho \\ \Rightarrow f_\beta(f\,x_1\,x_2 \ldots x_n) = f\,y_1\,y_2 \ldots y_n))\end{aligned}$$

which gives us a fusion law for fusing any function f_β with f.

Our previous analysis indicates that we should transform a function into a traversal scheme in such a way that the scheme's output type be completely parametric (a type variable). Having done this, we can easily generate the parametricity theorem for the scheme and use it to perform program fusion in the same way we use the fold fusion law to fuse a function with a fold.

The following is an example of a function that does not have a direct representation as a fold. We will transform it to get a completely polymorphic output. This function is zip[1]:

```
zip(a:x,b:y) = (a,b):(zip(x,y))
zip _        = [ ]
```

Its type is polymorphic:

$$\forall\alpha,\beta.\ (\text{list}(\alpha) \times \text{list}(\beta)) \rightarrow \text{list}(\alpha \times \beta)$$

but the parametricity theorem for this type is a simple natural transformation:

$$\forall f_\alpha, f_\beta :\ \text{map}(f_\alpha \times f_\beta) \circ \text{zip} \ = \ \text{zip} \circ (\text{map}(f_\alpha) \times \text{map}(f_\beta))$$

Notice that the output type of zip is not a type variable. Thus, this theorem cannot be used as is for fusing any function g with zip. To make the output type of zip a type variable, we should generalize both the inductive equations of zip. First observe that the second equation returns [ ]; this should be replaced by an extra parameter, n, of zip. Finally, the first equation returns a list construction; this too should be abstracted into another extra parameter, c. The transformed function zip is now:

```
zip'(c,n)(a:x,b:y) = c(a,b,zip'(c,n)(x,y))
zip'(c,n) _        = n()
```

which has a sufficiently polymorphic type:

$$\forall\alpha,\beta,\gamma.\ ((\alpha \times \beta \times \gamma \rightarrow \gamma) \times (() \rightarrow \gamma)) \rightarrow (\text{list}(\alpha) \times \text{list}(\beta)) \rightarrow \gamma$$

since its output type is the type variable γ. Function zip can be computed in terms of zip':

```
zip = zip'( λ(a,b,r). (a,b):r, λ(). [ ] )
```

In a way, zip' is a worker and the above definition of zip is a wrapper [8]. The parametricity theorem for zip' with $\alpha = \beta = ()$ is:

$$\forall f_\gamma, c, n, c' :\ f_\gamma \circ c = c' \circ (\text{id} \times \text{id} \times f_\gamma)\ \Rightarrow\ f_\gamma \circ \text{zip}'(c, n) = \text{zip}'(c', f_\gamma \circ n)$$

Suppose now that we want to perform the program fusion len(zip(x,y)). We can achieve this fusion by unwrapping zip and by using the zip' fusion law for f_γ = len and c(a,b,r)=(a,b):r:

[1]Function zip can be expressed as a second-order fold that traverses one of the zip arguments and deconstructs the other argument during the traversal. This results into an asymmetry: the fold fusion law can only be used for fusing one argument only. An alternative, symmetric, definition of zip is given elsewhere [4] but it requires a more general traversal scheme than fold.

```
len(zip(x,y)) = len( zip'( λ(a,b,r). (a,b):r, λ(). [ ] )(x,y) )
              = zip'( c', λ(). len [ ] )(x,y)
              = zip'( c', λ(). 0 )(x,y)
```

The premise of the zip' fusion law gives us a value for c':

```
c'(x,y,len z) = len(c(x,y,z))
              = len((x,z):z)
              = 1+(len z)
```

Therefore, if we generalize the term len z to a variable w, we get c'(x,y,w) = 1+w. That is,

```
len(zip(x,y)) = zip'( λ(x,y,w). 1+w, λ(). 0 )(x,y)
```

Finally, if we unroll zip' and set f = zip'(λ(x,y,w). 1+w, λ(). 0), we get:

```
f(a:x,b:y) = 1+(f(x,y))
f _        = 0
```

5 The Fusion Algorithm

Program fusion in our framework is perform in five steps:

- Given a function f, generate a sufficiently polymorphic function $\mathcal{SK}_f$, called the *recursive skeleton* of f, that captures the recursion scheme of f;
- Redefine f as the wrapper of $\mathcal{SK}_f$, i.e., $f = \mathcal{SK}_f(e_1, \ldots, e_n)$, for some expressions e_i;
- Generate the parametricity theorem for the type of $\mathcal{SK}_f$, with $\alpha = ()$ for any type variable α other than the type variable of the output;
- Whenever there is an application $g(f\,e)$ in a program, unwrap f into $\mathcal{SK}_f$ and use the parametricity theorem to fuse the application.

These steps are described in greater detail below.

5.1 Extracting the Recursive Skeleton of a Function

This section presents an algorithm for extracting the recursive skeleton of a function f of type $t_1 \to t_2 \to \cdots t_n$. It works over functions f defined in terms of m recursive equations:

$$\begin{array}{rcl} f\,p_{1,1} \cdots p_{1,n} & = & e_1 \\ \vdots & & \\ f\,p_{m,1} \cdots p_{m,n} & = & e_m \end{array}$$

where $p_{i,j}$ is a pattern and e_i is an expression that may contain recursive calls to f. Each such recursive call must provide n arguments to f, i.e., $f\,a_1 \cdots a_n$, and each a_i is an expression whose free variables are bound exclusively in the patterns $p_{i,j}$. In that case, the skeleton of f is $\mathcal{SK}_f$, whose i*th* inductive equation is defined as follows:

$$\mathcal{SK}_f(g_1,\ldots,g_m)\,p_{i,1}\cdots p_{i,n} \;=\; g_i(\overline{v_i},\overline{\mathcal{SK}_f(g_1,\ldots,g_m)\,a_{i,1}\cdots a_{i,n}})$$

That is, we assign a function g_i for the output of each inductive equation and we collect all variables $\overline{v_i}$ that appear in the patterns and all the recursive calls, $\mathcal{SK}_f(g_1,\ldots,g_m)\,a_{i,1}\cdots a_{i,n}$, in e_i as arguments to g_i. Function f can be defined in terms of $\mathcal{SK}_f$ by expressing g_i as follows:

$$g_i \;=\; \lambda(\overline{v_i},\overline{r_i}).\, e_i[\overline{r_i/\mathcal{SK}_f(g_1,\ldots,g_m)\,a_{i,1}\cdots a_{i,n}}]$$

that is, g_i is equal to e_i with variable r_j substituted for each recursive call.

The type of $\mathcal{SK}_f$ is (when we set all but the output type variable to ()):

$$\forall\alpha.\;(\tau_1\rightarrow\alpha)\times\cdots\times(\tau_m\rightarrow\alpha)\rightarrow t_1\rightarrow\cdots\rightarrow t_n\rightarrow\alpha$$

The parametricity theorem for this type is:

$$\forall f_\alpha, g_i, g_i', x_i: \quad \bigwedge_i f_\alpha\circ g_i = g_i'\circ\mathcal{F}[\![\tau_i]\!](f_\alpha,\mathrm{id}) \;\Rightarrow\; f_\alpha(\mathcal{SK}_f(\overline{g})\,\overline{x}) = \mathcal{SK}_f(\overline{g'})\,\overline{x} \qquad (5)$$

Proof: The parametricity theorem for the type of $\mathcal{SK}_f$ is $\bigwedge_i \mathcal{P}[\![\tau_i\rightarrow\alpha]\!](g_i',g_i)\,\rho \Rightarrow f_\alpha(\mathcal{SK}_f(\overline{g})\,\overline{x}) = \mathcal{SK}_f(\overline{g'})\,\overline{x}$. We have:

$$\begin{array}{rcll}
\mathcal{P}[\![\tau_i\rightarrow\alpha]\!](g_i',g_i)\,\rho & \Leftrightarrow & \mathcal{P}[\![\tau_i]\!](x,y)\,\rho\Rightarrow g_i'(x)=f_\alpha(g_i(y)) & \text{from Th. 1}\\
 & \Leftrightarrow & x=\mathcal{F}[\![\tau_i]\!](f_\alpha,\mathrm{id})\,y\Rightarrow g_i'(x)=f_\alpha(g_i(y)) & \text{from Th. 4}\\
 & \Leftrightarrow & g_i'(\mathcal{F}[\![\tau_i]\!](f_\alpha,\mathrm{id})\,y)=f_\alpha(g_i(y)) & \Box
\end{array}$$

For example, consider the list reverse function:

```
rev [ ]    = [ ]
rev (a:x)= append (rev x) [a]
```

Its recursive skeleton $\mathcal{SK}_{\mathsf{rev}}$ = rev' is straightforward:

```
rev'(n,c) [ ]    = n()
rev'(n,c) (a:x)= c(a,x,rev'(n,c) x)
```

which is actually equivalent to the list primitive recursion. Function rev can be expressed in terms of rev':

$$\mathsf{rev} = \mathsf{rev'}(\;\lambda().\;[\,],\;\lambda(\mathsf{a,x,r}).\;\mathsf{append\;r\;[a]}\;)$$

The type of rev' is:

$$\forall\alpha,\beta.\;(()\rightarrow\alpha)\times(\beta\times\mathrm{list}(\beta)\times\alpha\rightarrow\alpha)\rightarrow\mathrm{list}(\beta)\rightarrow\alpha$$

which satisfies the following parametricity theorem (for $\beta = ()$):

$$\forall f_\alpha, n, c, n', c', x:\; f_\alpha\circ n = n'\circ\mathrm{id}\;\wedge\; f_\alpha\circ c = c'\circ(\mathrm{id}\times\mathrm{id}\times f_\alpha)\;\Rightarrow\; f_\alpha(\mathrm{rev}'(n,c)\,x) = \mathrm{rev}'(n',c')x$$

The above algorithm can be easily extended to allow recursive calls $f\,a_1 \cdots a_n$ in which some variables in a_i are bound in an outer case statement. In that case, we would need more extra parameters for $\mathcal{SK}_f$; one for each case branch. A more substantial extension can be achieved by permitting a_i to contain variables that do not appear in the function patterns or in the outer case statements. In that case we do not abstract the function call but instead we construct a lambda expression that captures all these free variables. If all arguments to the recursive calls are free variables, our method deteriorates to the unfold-simplify-fold law (Eq. 2), because the recursive skeleton becomes the Y combinator. Since this is undesirable, we try to abstract as many arguments of the recursive calls as possible.

For example, consider the following program that computes the map over bushes:

$$\begin{aligned}&\mathsf{mapB(f)(Leaf\ x)} &&= \mathsf{Leaf(f\ x)}\\ &\mathsf{mapB(f)(Branch\ r)} &&= \mathsf{Branch(map(\lambda z.\ mapB(f)\ z)\ r)}\end{aligned}$$

where, Leaf and Branch are the value constructors of Bush:

$$\mathsf{data\ Bush(\alpha) = Leaf\ \alpha \mid Branch\ list(Bush(\alpha))}$$

Notice that the variable z in the second equation is not bound in the pattern of the equation. Thus, in this case we do not abstract the recursive call alone, but a lambda abstraction that contains the recursive call:

$$\begin{aligned}&\mathsf{mapB'(l,b)(f)(Leaf\ x)} &&= \mathsf{l(f,x)}\\ &\mathsf{mapB'(l,b)(f)(Branch\ r)} &&= \mathsf{b(r,\lambda z.\ mapB'(l,b)(f)\ z)}\end{aligned}$$

Function mapB is defined in terms of mapB':

$$\mathsf{mapB = mapB'(\ \lambda(f,x).\ Leaf(f\ x),\ \lambda(r,g).\ Branch(map(g)\ r)\ )}$$

The type of mapB' is:

$$\forall \alpha,\beta,\gamma.\ (\alpha \times \beta \rightarrow \gamma) \times ((\mathrm{list}(\mathrm{Bush}(\beta)) \times (\mathrm{Bush}(\beta) \rightarrow \gamma)) \rightarrow \gamma) \rightarrow \alpha \rightarrow \mathrm{Bush}(\beta) \rightarrow \gamma$$

The parametricity theorem for $\alpha = \beta = ()$ is:

$$\begin{aligned}&\forall f_\gamma, l, b, b', f, x:\ f_\gamma \circ b = b' \circ (\mathrm{id} \times (\lambda g.\, f_\gamma \circ g))\\ &\qquad \Rightarrow\ f_\gamma(\mathrm{mapB}'(l,b)(f)\,x) = \mathrm{mapB}'(f_\gamma \circ l, b')(f)\,x\end{aligned}$$

5.2 The Fusion Algorithm

The problem of fusing two recursive functions f and h in $h(f(x))$ is to derive a new recursive function with the same functionality as $h(f(x))$. This is not always possible. Typically, f produces an intermediate data structure which is consumed by h. When these two functions are fused, this data structure is not generated. In our framework, the fusion $h(f(x))$ is achieved by fusing $h(\mathcal{SK}_f(\overline{g})\,x)$, since $f = \mathcal{SK}_f(\overline{g})\,x)$, for some functions g_i. The law for this fusion is derived directly from the parametricity theorem of $\mathcal{SK}_f$.

We have seen that even with all the extensions described in Section 5.1, the type of $\mathcal{SK}_f$ has the following form:

$$\forall\alpha.\ (\tau_1 \to \alpha) \times \cdots \times (\tau_m \to \alpha) \to t_1 \to \cdots \to t_n \to \alpha$$

Note that, type τ_i does not contain any negative instances of α. As we have seen from Eq. 5, the parametricity theorem for this type is:

$$\forall h, g_i, g_i', x_i : \quad \bigwedge_i h \circ g_i = g_i' \circ \mathcal{F}[\![\tau_i]\!](h, \mathrm{id}) \ \Rightarrow\ h(\mathcal{SK}_f(\overline{g})\,\overline{x}) = \mathcal{SK}_f(\overline{g'})\,\overline{x} \qquad (6)$$

This gives us a law for fusing any function h with f: h is given, each g_i is derived directly from the definition of f, and each $\mathcal{F}[\![\tau_i]\!]$ is derived from Definition 1. The only things that need to be computed are the g_i' functions.

In a previous work [9], we describe a method for solving a similar set of equations for fusing folds. It relies on the fact that for each function h there exists a function $\mathcal{INV}(h)$ such that $\forall x \in \mathrm{range}(h) :\ h(\mathcal{INV}(h)\,x) = x$, i.e., $\mathcal{INV}(h)$ is a right inverse of h. To see why this is true, consider all the values $a_1, \ldots, a_n$, $n > 0$ that satisfy $h(a_1) = \cdots = h(a_n) = b$ for some value $b \in \mathrm{range}(h)$. Then, $\mathcal{INV}(h)\,b$ is defined to be one of these a_i. The method described in [9] and also used in this paper, does not actually derive a function $\mathcal{INV}(h)$ from h. It only assumes that such function exists and uses the property $h \circ \mathcal{INV}(h) = \mathrm{id}$ to eliminate it.

Using $\mathcal{INV}(h)$, the premise of Eq. 6 becomes:

$$\begin{array}{lll}
& h \circ g_i = g_i' \circ \mathcal{F}[\![\tau_i]\!](h, \mathrm{id}) & \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](h, \mathrm{id}) \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) & \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](h \circ \mathcal{INV}(h), \mathrm{id}) & \text{from Eq. 4} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](\mathrm{id}, \mathrm{id}) & \text{since } h \circ \mathcal{INV}(h) = \mathrm{id} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathrm{id} & \text{from Eq. 3} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' &
\end{array}$$

Thus, we derive the following algorithm from Eq. 6:

Algorithm 1 (Fusion Algorithm)

$$\forall h, g_i, x_i :\ h(\mathcal{SK}_f(\overline{g})\,\overline{x}) \ \to\ \mathcal{SK}_f(\overline{g'})\,\overline{x} \qquad \text{where } g_i' = h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) \qquad (7)$$

$$\forall x :\ h(\mathcal{INV}(h)\,x) \ \to\ x \qquad (8)$$

Eq. 7 fuses h with f into a function that has the same recursive skeleton as f. The components g_i' of the resulting skeleton are derived by fusing h, g_i, and $\mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id})$. Since $\mathcal{INV}(h)$ is unknown, we only fuse h with g_i. This fusion can be achieved by using Eq. 7, Eq. 8, as well some standard partial evaluation techniques (such as applying a function to a value construction). At the end we hope to derive terms of the form $h \circ \mathcal{INV}(h)$ only, which cancel $\mathcal{INV}(h)$. Otherwise, if there is a term $\mathcal{INV}(h)$ in the program that is not cancelled out, then the fusion algorithm fails and the fusion $h(f\,x)$ is not performed.

As an example, we will fuse `nth (zip(x,y)) n`, where

```
nth [ ] n     = error()
nth (a:x) n   = if n=0 then a else nth x (n-1)
```

Using the fusion algorithm we get:

```
nth (zip(x,y)) n = nth (zip'( λ(a,b,r). (a,b):r, λ(). [ ] )(x,y)) n
                 = zip'( λ(a,b,r). nth ((a,b):(INV(nth) r)), λ(). nth [ ] )(x,y) n
                 = zip'( λ(a,b,r). λn. if n=0 then (a,b) else nth(INV(nth) r) (n-1),
                         λ(). λn. error() )(x,y) n
                 = zip'( λ(a,b,r). λn. if n=0 then (a,b) else r(n-1), λ(). λn. error() )(x,y) n
```

If we unroll zip' and set f (x,y) n = nth (zip(x,y)) n, we get:

```
f (a:x,b:y) n = if n=0 then (a,b) else f (x,y) (n-1)
f _ n         = error()
```

But there are examples in which $\mathcal{INV}(h)$ is not cancelled out, such as in the case of the length of the quadratic reverse. Using the fusion algorithm we get:

```
len(rev(x)) = len(rev'( λ(). [ ], λ(a,x,r). append r [a] ) x)
            = rev'( λ(). len([ ]), λ(a,x,r). len(append (INV(len) r) [a]) ) x
            = rev'( λ(). 0, λ(a,x,r). len(append (INV(len) r) [a]) ) x
```

The skeleton of append is:

```
append'(f,g) [ ] y    = f(y)
append'(f,g) (a:x) y  = g(a,x,y,append'(f,g) x y)
```

If we apply the fusion algorithm recursively to fuse len with append, we get:

```
len(append (INV(len) r) [a])
= len( append'( λy. y, λ(b,x,y,s). b:s ) (INV(len) r) [a] )
= append'( λy. len(y), λ(b,x,y,s). len(b:(INV(len) s)) ) (INV(len) r) [a]
= append'( λy. len(y), λ(b,x,y,s). 1+s ) (INV(len) r) [a]
```

Therefore, len(rev(x)) is

```
rev'( λ(). 0, λ(a,x,r). append'( λy. len(y), λ(b,x,y,s). 1+s ) (INV(len) r) [a] ) x
```

which contains a term $\mathcal{INV}$(len) that has not been cancelled out. This problem occurs whenever the result of a recursive call of a function is handled by another recursive function in a non-trivial way (i.e., when it is deconstructed and/or recursed upon it). All the other cases can be handled effectively by the fusion algorithm.

There are two ways to handle cases like this: One is to actually synthesize $\mathcal{INV}(h)$ from h. For example,

```
INV(len) = int_fold( λ().[ ], λn. 0:n )
```

where int_fold is the fold over integers. That way, we can fuse append' with $\mathcal{INV}$(len) yielding:

```
len(rev(x)) = rev'( λ(). 0, λ(a,x,r). int_fold( λ(). λy. len(y), λf. λy. 1+f(y) ) r [a] ) x
```

The other way is to use additional laws. These laws, called *promotion laws*, instead of fusing g with f in $g(f(x))$, they promote g to the right of f. That is, these laws take the form: $g \circ f = h \circ F(g)$, for some function h that does not depend on g and some function F that depends on g. For example, we have the law:

$$\mathsf{len(append\ x\ y) = (len(x)){+}(len(y))}$$

which actually states that len is a homomorphism. That way, len(rev(x)) becomes:

$$\begin{array}{l} \mathsf{rev'(\ \lambda().\ 0,\ \lambda(a,x,r).\ len(append\ (}\mathcal{INV}\mathsf{(len)\ r)\ [a])\ )\ x} \\ \mathsf{=rev'(\ \lambda().\ 0,\ \lambda(a,x,r).\ (len(}\mathcal{INV}\mathsf{(len)\ r)){+}(len([a]))\ )\ x} \\ \mathsf{=rev'(\ \lambda().\ 0,\ \lambda(a,x,r).\ r{+}1\ )\ x} \end{array}$$

In general, we need to solve the following equation to synthesize k:

$$\mathcal{SK}_f(\overline{g}) \circ \mathcal{SK}_f(\overline{h}) \;=\; k \circ \mathcal{SK}_f(\overline{g})$$

given $\mathcal{SK}_f$, g_i, and h_i. This equation may not have a solution in general.

6 Extensions

From Th. 3 we have:

$$\forall f_i, x, y: \quad \mathcal{P}[\![\tau]\!](x,y)[\overline{f_i/\alpha_i}] \quad \Leftrightarrow \quad x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{\mathcal{INV}(f)})\, y$$

since $f_i \circ \mathcal{INV}(f_i) = \mathrm{id}$. The fusion algorithm can be extended accordingly to handle any function $f : \tau_1 \to \cdots \to \tau_n \to \alpha$:

$$\begin{array}{rll} \forall h, x_i: & h(f\, x_1 \ldots x_n) & \to \; f\,(\mathcal{F}[\![\tau_1]\!](h, \mathcal{INV}(h))\, x_1) \cdots (\mathcal{F}[\![\tau_n]\!](h, \mathcal{INV}(h))\, x_n) \\ \forall x: & h(\mathcal{INV}(h)\, x) & \to \; x \end{array}$$

Using these extensions, we can achieve various types of fusion and deforestation, not captured by the regular fusion algorithm. The following sections describe some of them.

6.1 Accumulator Deforestation

The linear version of list reverse is rev x = reverse x [ ], where reverse uses an extra accumulator:

```
reverse [ ] w     = w
reverse (a:x) w = reverse x (a:w)
```

Its recursive skeleton is:

```
reverse'(f,g) [ ] w     = f(w)
reverse'(f,g) (a:x) w = g(reverse'(f,g) x (a:w))
```

If we use the regular fusion algorithm, we derive:

```
len(reverse x [ ]) = reverse'(len,id) x [ ]
```

which uses the accumulator in the same way as reverse does: it starts with an empty list and at each step it conses an element to the accumulator. At the end, it returns the length of the accumulator. We can avoid building the list accumulator by using a simple integer accumulator instead:

```
h [ ] w     = w
h (a:x) w = h x (1+w)
```

Such a definition can be derived by abstracting the accumulator from each recursive call in reverse':

```
reverse'(f,g) [ ] w     = f(w)
reverse'(f,g) (a:x) w = g(a,w,λz. reverse'(f,g) x z)
```

As before, reverse can be expressed in terms of reverse':

```
reverse = reverse'( λz.z, λ(a,w,f). f(a:w) )
```

The type of reverse' is:

$$\forall\alpha,\beta,\gamma.\ (\gamma\rightarrow\beta)\times(\alpha\times\gamma\times(\gamma\rightarrow\beta)\rightarrow\beta)\rightarrow\text{list}(\alpha)\rightarrow\gamma\rightarrow\beta$$

If we set $\alpha = ()$ and $\beta = \gamma$, then the extended fusion algorithm gives:

$$h(\mathsf{reverse'}(f,g)\ x\ w) \quad\rightarrow\quad \mathsf{reverse'}(h\circ f\circ\mathcal{INV}(h), h\circ g\circ(\text{id}\times\mathcal{INV}(h)\times(\lambda k.\,\mathcal{INV}(h)\circ k\circ h)))\ x\ (h(w))$$

Using this rule and after some simplifications, we get:

```
len(reverse x [ ]) = reverse'( λz.z, λ(a,w,f). f(1+w) ) x 0
```

which is equivalent to the desired definition of len(rev x).

References

[1] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[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] L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, January 1996. To Appear.

[4] L. Fegaras, T. Sheard, and T. Zhou. Improving Programs which Recurse over Multiple Inductive Structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, pp 21–32, June 1994.

[5] 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.

[6] J. Launchbury and T. Sheard. Warm Fusion. *Seventh Conference on Functional Programming Languages and Computer Architecture, La Jolla, California*, pp 314–323, June 1995.

[7] 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. Springer-Verlag, LNCS 523, August 1991.

[8] S. Peyton Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-strict Functional Language. *Fifth Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pp 636–665, August 1991.

[9] 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.

[10] P. Wadler. Theorems for Free! *Fourth Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, September 1989.

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

A Proofs

Theorem 3 *For any type $\forall \alpha_1, \ldots, a_n : \tau$ we have:*

$$\forall f_i, g_i, x, y : \quad f_i \circ g_i = \mathrm{id} \quad \Rightarrow \quad (\mathcal{P}[\![\tau]\!](x,y)\overline{[f_i/\alpha_i]} \quad \Leftrightarrow \quad x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{g})\, y)$$

Proof (by induction over the structure of the type τ):

- If $\tau = $ basic, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) = \mathrm{id}$ and $\mathcal{P}[\![\tau]\!](x,y)\overline{[f_i/\alpha_i]} \Leftrightarrow (x = y)$.
- If $\tau = \alpha_i$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) = f_i$ and $\mathcal{P}[\![\tau]\!](x,y)\overline{[f_i/\alpha_i]} \Leftrightarrow (x = f_i(y))$.
- If $\tau = \tau_1 \times \tau_2$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) = \mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g}) \times \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g})$, and

$$\begin{array}{lll} & \mathcal{P}[\![\tau]\!](x,y)\overline{[f_i/\alpha_i]} & \\ \Leftrightarrow & \mathcal{P}[\![\tau_1]\!](\pi_1(x), \pi_1(y))\overline{[f_i/\alpha_i]} \wedge \mathcal{P}[\![\tau_2]\!](\pi_2(x), \pi_2(y))\overline{[f_i/\alpha_i]} & \text{from Th. 1} \\ \Leftrightarrow & (\pi_1(x) = \mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g})\ (\pi_1(y))) \wedge (\pi_2(x) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g})\ (\pi_2(y))) & \text{induction hypothesis} \\ \Leftrightarrow & x = (\mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g}) \times \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}))\ y & \\ \Leftrightarrow & x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{g})\ y & \end{array}$$

- If $\tau = \tau_1 \rightarrow \tau_2$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) = \lambda h. \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f})$, and

$$\begin{array}{lll} & \mathcal{P}[\![\tau]\!](k,h)\overline{[f_i/\alpha_i]} & \\ \Leftrightarrow & \mathcal{P}[\![\tau_1]\!](x,y)\overline{[f_i/\alpha_i]} \Rightarrow \mathcal{P}[\![\tau_2]\!](k(x), h(y))\overline{[f_i/\alpha_i]} & \text{from Th. 1} \\ \Leftrightarrow & x = \mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g})\ y \Rightarrow k(x) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g})\ (h(y)) & \text{induction hypothesis} \\ \Leftrightarrow & k \circ \mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g}) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h & \\ \Leftrightarrow & k \circ \mathcal{F}[\![\tau_1]\!](\overline{f}, \overline{g}) \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f}) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f}) & (*) \\ \Leftrightarrow & k \circ \mathcal{F}[\![\tau_1]\!](\overline{f \circ g}, \overline{f \circ g}) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f}) & \text{from Eq. 4} \\ \Leftrightarrow & k \circ \mathcal{F}[\![\tau_1]\!](\overline{\mathrm{id}}, \overline{\mathrm{id}}) = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f}) & \text{since } f_i \circ g_i = \mathrm{id} \\ \Leftrightarrow & k \circ \mathrm{id} = \mathcal{F}[\![\tau_2]\!](\overline{f}, \overline{g}) \circ h \circ \mathcal{F}[\![\tau_1]\!](\overline{g}, \overline{f}) & \text{from Eq. 3} \\ \Leftrightarrow & k = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{g})\ h & \end{array}$$

  The equivalence (*) is true because $\mathrm{range}(h) = \mathrm{domain}(f) \ \Rightarrow \ (f = g \Leftrightarrow f \circ h = g \circ h)$.

- If $\tau = T(\tau')$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) = \mathrm{map}^T(\mathcal{F}[\![\tau']\!](\overline{f}, \overline{g}))$, and

$$\begin{array}{lll} & \mathcal{P}[\![\tau]\!](x,y)\overline{[f_i/\alpha_i]} & \\ \Leftrightarrow & \forall h : (\forall z : \mathcal{P}[\![\tau']\!](h(z), z)\overline{[f_i/\alpha_i]}) \Rightarrow x = \mathrm{map}^T(h)\ y & \text{from Th. 1} \\ \Leftrightarrow & \forall h : (\forall z : h(z) = \mathcal{F}[\![\tau']\!](\overline{f}, \overline{g})\ z) \Rightarrow x = \mathrm{map}^T(h)\ y & \text{induction hypothesis} \\ \Leftrightarrow & x = \mathrm{map}^T(\mathcal{F}[\![\tau']\!](\overline{f}, \overline{g}))\ y & \\ \Leftrightarrow & x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{g})\ y & \square \end{array}$$