A Proofs

Theorem 4 *For any type* $\forall\alpha_1,\ldots,a_n:\ \tau,\ n\geq 0$ *we have:*

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

where $\overline{f} = f_{\alpha_1},\ldots,f_{\alpha_n}$ *and* $\overline{g} = g_{\alpha_1},\ldots,g_{\alpha_n}$.

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

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

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

- If $\tau = \tau_1\to\tau_2$, 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

$$\begin{array}{lll}
 & \mathcal{P}[\![\tau]\!](k,h) & \\
\Leftrightarrow & \mathcal{P}[\![\tau_1]\!](x,y) \Rightarrow \mathcal{P}[\![\tau_2]\!](k(x),h(y)) & \text{from Th. 1} \\
\Leftrightarrow & x = \mathcal{F}[\![\tau_1]\!](\overline{f},\overline{g})\, y \Rightarrow k(x) = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\,(h(y)) & \text{induction hypothesis} \\
\Leftrightarrow & k\circ\mathcal{F}[\![\tau_1]\!](\overline{f},\overline{g}) = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h & \\
\Leftrightarrow & k\circ\mathcal{F}[\![\tau_1]\!](\overline{f},\overline{g})\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f}) = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f}) & (*) \\
\Leftrightarrow & k\circ\mathcal{F}[\![\tau_1]\!](\overline{f\circ g},\overline{f\circ g}) = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f}) & \text{from Eq. 5} \\
\Leftrightarrow & k\circ\mathcal{F}[\![\tau_1]\!](\overline{\mathrm{id}},\overline{\mathrm{id}}) = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f}) & \text{since } f_i\circ g_i = \mathrm{id} \\
\Leftrightarrow & k\circ\mathrm{id} = \mathcal{F}[\![\tau_2]\!](\overline{f},\overline{g})\circ h\circ\mathcal{F}[\![\tau_1]\!](\overline{g},\overline{f}) & \text{from Eq. 4} \\
\Leftrightarrow & k = \mathcal{F}[\![\tau]\!](\overline{f},\overline{g})\, h &
\end{array}$$

  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

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

[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] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96), Philadelphia*, pp 73–82. ACM Press, May 1996.

[7] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiple Data Traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pp 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

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

[9] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144, August 1991. LNCS 523.

[10] 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.

[11] 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.

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

[13] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, pp 344–358, March 1988. LNCS 760.

$$\begin{array}{l} \mathsf{reverse'(f,g)\ [\ ]\ w} \quad = \mathsf{f(w)} \\ \mathsf{reverse'(f,g)\ (a:x)\ w} = \mathsf{g(reverse'(f,g)\ x\ (a:w))} \end{array}$$

If we use the regular fusion algorithm, we derive:

$$\mathsf{len(reverse\ x\ [\ ])} = \mathsf{reverse'(len,id)\ x\ [\ ]}$$

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:

$$\begin{array}{l} \mathsf{h\ [\ ]\ w} \quad = \mathsf{w} \\ \mathsf{h\ (a:x)\ w} = \mathsf{h\ x\ (1+w)} \end{array}$$

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

$$\begin{array}{l} \mathsf{reverse'(f,g)\ [\ ]\ w} \quad = \mathsf{f(w)} \\ \mathsf{reverse'(f,g)\ (a:x)\ w} = \mathsf{g(a,w,}\lambda\mathsf{z.\ reverse'(f,g)\ x\ z)} \end{array}$$

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

$$\mathsf{reverse} = \mathsf{reverse'(\ }\lambda\mathsf{z.z,\ }\lambda\mathsf{(a,w,f).\ f(a:w)\ )}$$

The type of **reverse'** is:

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

From Th. 4 we have $\forall f_{\alpha_i}, x, y:\ \mathcal{P}[\![\tau]\!](x,y) \ \Leftrightarrow\ x = \mathcal{F}[\![\tau]\!](\overline{f}, \overline{\mathcal{INV}(f)})\, y$, since $f_i \circ \mathcal{INV}(f_i) = \mathrm{id}$. The fusion algorithm can be extended accordingly to handle any function $f:\tau_1\to\cdots\to\tau_n\to\alpha$:

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

Therefore, for $\alpha = ()$ and $\beta = \gamma$, the **reverse'** fusion rule is:

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

Using this rule and after some simplifications, we get:

$$\mathsf{len(reverse\ x\ [\ ])} = \mathsf{reverse'(\ }\lambda\mathsf{z.z,\ }\lambda\mathsf{(a,w,f).\ f(1+w)\ )\ x\ 0}$$

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.

$$\begin{array}{l}\mathsf{append'(f,g)\ [\ ]\ y} \quad = \mathsf{f(y)} \\ \mathsf{append'(f,g)\ (a:x)\ y} \ = \mathsf{g(a,x,y,append'(f,g)\ x\ y)}\end{array}$$

If we apply the fusion algorithm once again to fuse **len** with **append**, we get:

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

Therefore, **len(rev(x))** is

$$\mathsf{rev'(\ }\lambda\mathsf{().\ 0,\ }\lambda\mathsf{(a,x,r).\ append'(\ }\lambda\mathsf{y.\ len(y),\ }\lambda\mathsf{(b,x,y,s).\ 1+s\ )\ (}\mathcal{INV}\mathsf{(len)\ r)\ [a]\ )\ x}$$

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). In that case, the intermediate data structure has a true computational value in the program and cannot be eliminated. 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. We may have many solutions for $\mathcal{INV}(h)$ that satisfy $f \circ \mathcal{INV}(h) = \mathrm{id}$. One solution is

$$\mathcal{INV}\mathsf{(len)} = \mathsf{int\_fold(\ }\lambda\mathsf{().[\ ],\ }\lambda\mathsf{n.\ 0:n\ )}$$

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

$$\mathsf{len(rev(x))} = \mathsf{rev'(\ }\lambda\mathsf{().\ 0,\ }\lambda\mathsf{(a,x,r).\ int\_fold(\ }\lambda\mathsf{().\ }\lambda\mathsf{y.\ len(y),\ }\lambda\mathsf{f.\ }\lambda\mathsf{y.\ 1+f(y)\ )\ r\ [a]\ )\ x}$$

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:

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

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

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

4.4 Accumulator Deforestation

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

$$\begin{array}{l}\mathsf{reverse\ [\ ]\ w} \quad = \mathsf{w} \\ \mathsf{reverse\ (a:x)\ w} = \mathsf{reverse\ x\ (a:w)}\end{array}$$

Its recursive skeleton is:

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

$$
\begin{array}{lll}
& h \circ g_i = g_i' \circ \mathcal{F}[\![\tau_i]\!](h, \mathrm{id}) & \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](h, \mathrm{id}) \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) & \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](h \circ \mathcal{INV}(h), \mathrm{id}) & \text{from Eq. 5} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathcal{F}[\![\tau_i]\!](\mathrm{id}, \mathrm{id}) & \text{since } h \circ \mathcal{INV}(h) = \mathrm{id} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' \circ \mathrm{id} & \text{from Eq. 4} \\
\Leftrightarrow & h \circ g_i \circ \mathcal{F}[\![\tau_i]\!](\mathcal{INV}(h), \mathrm{id}) = g_i' &
\end{array}
$$

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

Algorithm 1 (Fusion Algorithm) *The fusion $h(f(\overline{x}))$ is achieved by the following rewrite rules:*

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

$$\forall x : \; h(\mathcal{INV}(h)\,x) \;\rightarrow\; x \tag{8}$$

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 $h(f(\overline{x}))$ 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 unfold zip', 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 $\mathcal{INV}(h)$ cannot 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:

Proof: Since all type variables α_i are positive in τ, none of the g_is in Th. 4 is used. Therefore, each g_i can be replaced by an arbitrary function, including the identity function itself. □

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

$$\mathcal{F}[\![\tau]\!](f_\alpha, g_\alpha) = \lambda h.\, (f_\alpha \times (\lambda k.\, \text{map}(f_\alpha) \circ k)) \circ h$$

We have $\mathcal{P}[\![\text{int} \rightarrow \text{list}(\alpha)]\!](g', g) \Leftrightarrow g' = \text{map}(f_\alpha) \circ g$ and

$$\begin{array}{rcl} \mathcal{P}[\![\tau]\!](h', h) & \Leftrightarrow & x = y \Rightarrow (\pi_1(h'(x)) = f_\alpha(\pi_1(h(y)))) \wedge (\pi_2(h'(x)) = \text{map}(f_\alpha) \circ (\pi_2(h(y)))) \\ & \Leftrightarrow & h'(x) = (f_\alpha \times (\lambda k.\, \text{map}(f_\alpha) \circ k))\,(h(x)) \end{array}$$

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

4.3 The Inverse Trick

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 $\mathcal{SK}_f$ given below.

We have seen that even with all the extensions described in Section 4.1, the type of $\mathcal{SK}_f$ has the following form:

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

Note that, type τ_i does not contain any negative instances of α. As we have seen from Eq. 3, the parametricity theorem for this type is:

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

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 functions that need to be computed are the g_i' functions.

In a previous work [11], 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 $\forall x \in \text{range}(h):\ h(\mathcal{INV}(h)\, x) = x$, 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 $b \in \text{range}(h)$. Then, $\mathcal{INV}(h)\, b$ is defined to be one of these a_i. This inverse trick, which is described in [11] and is 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 $h \circ \mathcal{INV}(h) = \text{id}$ to eliminate it.

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

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

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. The following theorem can be proved using structural induction over types:

Theorem 3 *If $f_i : \alpha_i^- \to \alpha_i^+$ and $g_i : \alpha_i^+ \to \alpha_i^-$, then $\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) : \tau^- \to \tau^+$.*

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 4 *For any type $\forall \alpha_1, \ldots, a_n : \tau$, $n \geq 0$ we have:*

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

where $\overline{f} = f_{\alpha_1}, \ldots, f_{\alpha_n}$ and $\overline{g} = g_{\alpha_1}, \ldots, g_{\alpha_n}$.

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

$$\begin{array}{lll}
\mathcal{P}[\![\tau]\!](x, y) & \Leftrightarrow & \forall m, n : m \circ f_\alpha = f_\beta \circ n \Rightarrow x(m) = f_\gamma(y(n)) \\
& \Leftrightarrow & \forall m, n : m \circ f_\alpha \circ g_\alpha = f_\beta \circ n \circ g_\alpha \Rightarrow x(m) = f_\gamma(y(n)) \qquad (\epsilon 1) \\
& \Leftrightarrow & \forall m, n : m = f_\beta \circ n \circ g_\alpha \Rightarrow x(m) = f_\gamma(y(n)) \\
& \Leftrightarrow & \forall n : x(f_\beta \circ n \circ g_\alpha) = f_\gamma(y(n)) \\
& \Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ n \circ g_\alpha) = f_\gamma \circ y \\
& \Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ n \circ g_\alpha) \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) \qquad (\epsilon 2) \\
& \Leftrightarrow & x \circ (\lambda n.\, f_\beta \circ g_\beta \circ n \circ f_\alpha \circ g_\alpha) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) \\
& \Leftrightarrow & x \circ (\lambda n.\, n) = f_\gamma \circ y \circ (\lambda n.\, g_\beta \circ n \circ f_\alpha) \\
& \Leftrightarrow & x = \mathcal{F}[\![\tau]\!](f_\alpha, f_\beta, f_\gamma, g_\alpha, g_\beta, g_\gamma)\, y
\end{array}$$

The equivalences ($\epsilon 1$) and ($\epsilon 2$) are based on the fact that $\mathrm{range}(h) = \mathrm{domain}(f) \;\Rightarrow\; (f = g \Leftrightarrow f \circ h = g \circ h)$.

Corollary 1 *Let $\forall \alpha_1, \ldots, a_n : \tau$, $n \geq 0$ be a type whose type variables are positive in τ, then:*

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

where $\overline{f} = f_{\alpha_1}, \ldots, f_{\alpha_n}$.

4.2 Expressing the Parametricity Theorem using Bifunctors

Most useful parametricity theorems are quite complex to use directly for program fusion since they typically include both a premise and a consequence. Thus, to fuse programs we need to synthesize a number of functions that satisfy the premise, which is hard in general. In this subsection we describe a theory for solving parametricity theorems effectively. A parametricity theorem relates two values of the same type. Here we prove that one value can be calculated in terms of the other under certain conditions. This calculation is based on the notion of the *bifunctor*. We will show in the next subsection that this expression of the parametricity theorem using bifunctors, even though it too has preconditions to be satisfied, these preconditions are eliminated if we use the *inverse trick* (to be described shortly), thus, calculating a solution directly.

A bifunctor is a generalization of a functor. In contrast to regular functors, bifunctors can capture types with contravariant type variables [3]. Here we define bifunctors only on a type whose universal quantifications appear in the outermost level, i.e., for types of the form $\forall \alpha_1, \ldots, a_n : \tau$ for $n \geq 0$.

Definition 1 (Bifunctor) *Let $\forall \alpha_1, \ldots, a_n : \tau$, $n \geq 0$ be a polymorphic type and let $\overline{f} = f_1, \ldots, f_n$ and $\overline{g} = g_1, \ldots, g_n$, where each f_i/g_i pair is associated with the type variable a_i. The* bifunctor *$\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g})$ is defined as follows:*

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

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,*

$$
\mathcal{F}[\![\tau]\!](\overline{\text{id}}, \overline{\text{id}}) \;=\; \text{id} \tag{4}
$$

$$
\mathcal{F}[\![\tau]\!](\overline{f}, \overline{g}) \circ \mathcal{F}[\![\tau]\!](\overline{f'}, \overline{g'}) \;=\; \mathcal{F}[\![\tau]\!](\overline{f \circ f'}, \overline{g' \circ g}) \tag{5}
$$

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

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

The type of rev' is:

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

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:

```
mapB(f)(Leaf x)   = Leaf(f x)
mapB(f)(Branch r)= Branch(map(λz. mapB(f) z) r)
```

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

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

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:

```
mapB'(l,b)(f)(Leaf x)   = l(f,x)
mapB'(l,b)(f)(Branch r)= b(r,λz. mapB'(l,b)(f) z)
```

Function mapB is defined in terms of mapB':

```
mapB = mapB'( λ(f,x). Leaf(f x), λ(r,g). Branch(map(g) r) )
```

The type of mapB' is:

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

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

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

4.1 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 can handle any function f defined in terms of m recursive equations:

$$\begin{array}{rcl} f\, p_{1,1} \cdots p_{1,n} & = & \epsilon_1(\overline{v_1}, \overline{f\, \epsilon_{1,1} \cdots \epsilon_{1,n}}) \\ \vdots & & \\ f\, p_{m,1} \cdots p_{m,n} & = & \epsilon_m(\overline{v_m}, \overline{f\, \epsilon_{m,1} \cdots \epsilon_{m,n}}) \end{array}$$

where $p_{i,j}$ is a pattern and ϵ_i is a function in which all variables $\overline{v_i}$ that appear in the patterns and all the recursive calls $f\, \epsilon_{i,1} \cdots \epsilon_{i,n}$ have been abstracted out of the body of ϵ_i. That is, the body of ϵ_i does not contain any free variable or any recursive calls to f. In that case, the skeleton of f is $\mathcal{SK}_f$, whose i*th* inductive equation is defined as follows:

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

Therefore, $f = \mathcal{SK}_f(\epsilon_1, \ldots, \epsilon_m)$.

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

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

The parametricity theorem for this type is:

$$\forall f_\alpha, g_i, g_i', x_i : \qquad \bigwedge_i f_\alpha \circ g_i = g_i' \circ \mathcal{F}[\![\tau_i]\!](f_\alpha, \mathrm{id}) \quad \Rightarrow \quad f_\alpha(\mathcal{SK}_f(\overline{g})\, \overline{x}) = \mathcal{SK}_f(\overline{g'})\, \overline{x} \tag{3}$$

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

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

For example, consider the list reverse function:

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

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

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

Following the terminology in [10], zip' can be viewed as a worker and the above definition of zip as a wrapper. The parametricity theorem for zip' with $\alpha = \beta = ()$ is:

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

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:

$$\begin{aligned}\mathsf{len(zip(x,y))} &= \mathsf{len(\ zip'(\ \lambda(a,b,r).\ (a,b):r,\ \lambda().\ [\,]\ )(x,y)\ )}\\ &= \mathsf{zip'(\ c',\ \lambda().\ len\ [\,]\ )(x,y)}\\ &= \mathsf{zip'(\ c',\ \lambda().\ 0\ )(x,y)}\end{aligned}$$

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

$$\begin{aligned}\mathsf{c'(x,y,len\ z)} &= \mathsf{len(c(x,y,z))}\\ &= \mathsf{len((x,z):z)}\\ &= \mathsf{1{+}(len\ z)}\end{aligned}$$

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

$$\mathsf{len(zip(x,y))} = \mathsf{zip'(\ \lambda(x,y,w).\ 1{+}w,\ \lambda().\ 0\ )(x,y)}$$

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

$$\begin{aligned}\mathsf{f(a{:}x,b{:}y)} &= \mathsf{1{+}(f(x,y))}\\ \mathsf{f\ \_} &= \mathsf{0}\end{aligned}$$

4 The Fusion Algorithm

Program fusion in our framework is perform in four steps:

- Given a function f, generate a sufficiently polymorphic function $\mathcal{SK}_f$, called the *recursive skeleton* of f, which captures the recursion scheme of f;
- Redefine f as a wrapper of $\mathcal{SK}_f$, i.e., $f = \mathcal{SK}_f(\epsilon_1, \ldots, \epsilon_n)$, for some expressions ϵ_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 $g(f\,\epsilon)$ in a program, unwrap f into an $\mathcal{SK}_f$ call and use the parametricity theorem to fuse the application.

As is it will be explained shortly, our fusion algorithm is based on a simple term rewriting system and does not involve theorem proving or program synthesis.

The above steps are described in greater detail below.

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

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

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 (i.e., 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 _        = [ ]
```

Even though, its type is polymorphic:

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

its parametricity theorem is a simple natural transformation:

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

Notice that the output type of **zip** is not a type variable. Thus, the above equation cannot be used as is for fusing any function g with **zip**. To make the output type of **zip** a type variable, we need to 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:

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

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(). [ ] )
```

[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.

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,

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

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 [8], 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 [9]. Recent work [6, 7] has been shown that most functional programs can be directly translated into hylomorphisms. It has yet to be shown the effectiveness of program fusion under this representation.

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** (Equation 1). 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 here):

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

theorem for $t(\alpha, \beta, \ldots)$ and then setting $f_\alpha = \mathrm{id}$ 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:

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

that satisfies for $\beta = ()$:

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

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:

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

which is actually the unfold-simplify-fold law [1] since it fuses any function f_β with a recursive function $Y\,f$ yielding another recursive function $Y\,f'$ (under certain conditions).

3 Using the Parametricity Theorem for Program Fusion

Consider the following non-polymorphic function of type list(int)→list(int) expressed in Haskell:

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

The parametricity theorem of a non-polymorphic type is always a tautology, which is of no use. 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 [11] 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 ⊕ r=(1+a):r, x=[ ], and y=x. From the conclusion of Eq. 1 we have:

```
len(inc x) = len(fold(λa. λr. (1+a):r) [ ] x)
           = fold(⊗) (len [ ]) x
           = fold(⊗) 0 x
```

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:

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

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.

The proof of the following theorem comes directly from [12] (here binary relations between values are restricted to value equalities):

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

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

We assume that each type variable, α, has a distinct name and is associated with a function f_α of type $\alpha_1 \to \alpha_2$, where α_1 and α_2 are instances of α.

For example, the list fold has type:

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

and satisfies the following parametricity theorem:

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

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

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

which is the fusion law for list fold [11]. Binding the function f_α, which corresponds to the type variable α, to id will be done very 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

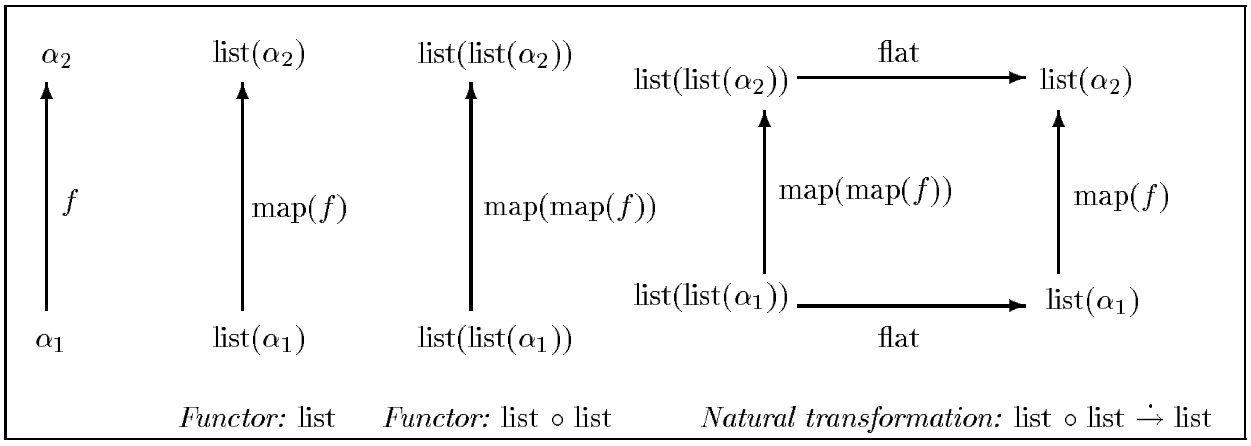

Figure 1: The Parametricity Theorem for flat : $\forall\alpha.\text{list}(\text{list}(\alpha))\rightarrow\text{list}(\alpha)$

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.

For a functional type, the parametricity theorem becomes a proposition of the form premise⇒conclusion. The conclusion can only be useful if the premise is proved first, a process that typically involves program synthesis. In this paper we report on a method of using parametricity theorems effectively for program fusion by expressing these theorems in terms of bifunctors. That way, program fusion is performed by calculating the fused program directly, without the need of theorem proving or program synthesis.

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* [12]), 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:

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

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 $\alpha_1\rightarrow\alpha_2$ into the function map(f) of type list(α_2)$\rightarrow$list(α_2). In general, any parametric

Using the Parametricity Theorem for Program Fusion

Leonidas Fegaras

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: *fegaras@cse.uta.edu*

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* [11] and *build* [8, 5], to automate program fusion [2] and deforestation [13]. 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 [8]. These methods had a moderate success so far, mostly because they can apply to simple functions only.

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 [12] 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, 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