JFP **13** (4): 797–814, July 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796802004409 Printed in the United Kingdom

Short cut fusion is correct

PATRICIA JOHANN
Department of Computer Science, Rutgers University, Camden, NJ 08102, USA
(*e-mail:* `pjohann@crab.rutgers.edu`)

Abstract

Fusion is the process of removing intermediate data structures from modularly constructed functional programs. *Short cut fusion* is a particular fusion technique which uses a single, local transformation rule to fuse compositions of list-processing functions. Short cut fusion has traditionally been treated purely syntactically, and justifications for it have appealed either to intuition or to "free theorems" – even though the latter have not been known to hold in languages supporting higher-order polymorphic functions and fixpoint recursion. In this paper we use Pitts' recent demonstration that contextual equivalence in such languages is parametric to provide the first formal proof of the correctness of short cut fusion for them. In particular, we show that programs which have undergone short cut fusion are contextually equivalent to their unfused counterparts.

Capsule Review

This paper proves the correctness of the local program transformation known as 'short cut fusion' for recursively defined, polymorphic functions processing lazy lists. It considerably improves upon previous results of this kind by considering operational (also known as contextual) equivalence of expressions in a language combining Plotkin's PCF with the Girard-Reynolds polymorphic lambda calculus and lazy lists: so we have fully impredicative polymorphism, higher order functions and fix point recursion, all given a non-strict operational semantics. (Details of how to generalise the correctness proof to cope with covariant recursive types more general than lazy lists appear elsewhere.) The proof of correctness relies on the operationally-based logical relations machinery developed by Pitts (Pitts, 2000).

1 Introduction

Modular program construction is widely regarded as an integral part of any reasonable software development process. One very general way of achieving modularity in functional languages is to construct large programs as compositions of small, generally applicable components. Each component in such a composition produces a data structure as its output, and this data structure is immediately consumed by the next component in the composition. *Intermediate data structures* thus serve as a kind of "glue" allowing components to be combined in a mix-and-match fashion.

The components comprising modular programs are typically defined as recursive functions. The definitions in figure 1 are common examples of such functions: `foldr`

```
foldr :: forall a. forall b. b -> (a -> b -> b) -> List a -> b
foldr = /\a b. \n c xs. case xs of
                          Nil -> n
                          Cons z zs -> c z (foldr a b n c zs)

map :: forall a. forall b. (a -> b) -> List a -> List b
map = /\a b. \f l. case l of
                     Nil -> Nil
                     Cons z zs -> Cons (f z) (map a b f zs)

append :: forall a. List a -> List a -> List a
append = /\a. \xs ys. case xs of
                        Nil -> ys
                        Cons z zs -> Cons z (append a zs ys)
```

Fig. 1. Recursive functions on lists.

consumes lists, while `map` and `append` both consume and produce lists. Using these functions we can define, for example, the function `mappend` which maps a function over the result of appending two lists:

```
mappend :: forall a. forall b. (a -> b) -> List a -> List a -> List b
mappend = /\a b. \f xs ys. map a b f (append a xs ys)
```

In this informal discussion we will express program fragments in a Haskell-like notation with explicit type quantification, abstraction, and application. Quantification of the type `t` over the type variable `a` is denoted `forall a.t`, abstraction of the term `M` over the type variable `a` is denoted `/\a.M`, and application of the term `M` to the type `t` is denoted `M t`.

Unfortunately, modularly constructed programs like `mappend` tend to be less efficient than their non-modular counterparts. The main difficulty is that the direct implementation of compositional programs *literally* constructs, traverses, and discards intermediate data structures – even when they play no essential role in a computation. For instance, the above implementation of `mappend` unnecessarily constructs and then traverses the intermediate list resulting from appending `xs` onto `ys`. This requires processing the list `xs` twice. Even in lazy languages this is expensive, both slowing execution time and increasing heap space requirements.

It is often possible to avoid manipulating intermediate data structures by using a more elaborate style of programming in which the computations performed by component functions in a composition are intermingled. In this monolithic style of programming the function `mappend` is defined as

```
mappend' :: forall a. forall b. (a -> b) -> List a -> List a -> List b
mappend' = /\a b. \f xs ys.
                 case xs of
                     Nil -> map a b f ys
                     Cons z zs -> Cons (f z) (mappend' a b f zs ys)
```

The list `xs` is only processed once by `mappend'`.

Experienced programmers writing a function to map over the result of appending two lists would instinctively produce `mappend'` rather than `mappend`; small functions like `mappend` are easily optimized at the keyboard. Because they are used very often, it is essential that small functions are optimized whenever possible. Automatic fusion tools ensure that they are.

On the other hand, when programs are either very large or very complex, even experienced programmers may find that eliminating intermediate data structures by hand is not a very attractive alternative to the modular style of programming. Methods for automatically eliminating intermediate data structures are needed in this situation as well.

1.1 Short cut fusion

Fusion is the process of removing intermediate data structures from modularly constructed functional programs. In recent years, a number of program fusion techniques have been developed (Gill *et al.*, 1993; Sheard & Fegaras, 1993; Takano & Meijer, 1995). *Short cut fusion* (Gill, 1996; Gill *et al.*, 1993) is a particular technique that uses a single, local transformation rule – called the `foldr-build` rule – to fuse compositions of list-processing functions via canned applications of traditional fold/unfold program transformation steps. To participate in short cut fusion, list-processing functions must be expressible in terms of the list-consuming function `foldr` and the list-producing function `build`.

Operationally, `foldr` takes as input types `t` and `t'`, a replacement term `n::t'` for `Nil`, a replacement term `c :: t -> t' -> t'` for `Cons`, and a list `xs` of type `List t`. It replaces all (fully applied) occurrences of `Cons` in `xs` by `c`, and the single (fully applied) occurrence of `Nil` in `xs` by `n`. The result is a value of type `t'`. The definition of `foldr` appears in figure 1.

The function `build`, on the other hand, takes as input a type `t` and a term `M` providing a type-independent template for constructing "abstract" lists with "elements" of type `t`. It instantiates all occurrences of the "abstract" list constructors which appear in the result list specified by `M` with the "concrete" list constructors `Nil` and `Cons`. The result is a list of elements of type `t`. More precisely, if `t` is a type and `M` is any term with type `forall a. a -> (t -> a -> a) -> a`, then

```
build t M = M (List t) Nil Cons
```

Compositions of list-consuming and -producing functions defined in terms of `foldr` and `build` can be fused via *short cut* fusion for lists: If `M` is a term with type `forall a. a -> (t -> a -> a) -> a`, then any occurrence of `foldr t t' n c (build t M)` in a program can be replaced by `M t' n c`. Short cut fusion makes sense intuitively: the result of a computation is the same regardless of whether the function `M` is first applied to `List t`, `Nil`, and `Cons` and then the latter are replaced in the resulting list by `n` and `c`, respectively, or the abstract constructors in (an appropriate instance of) `M` are replaced by `n` and `c`, respectively, directly.

Figure 2 shows the `build-foldr` forms of the functions `map` and `append` from figure 1. The fused function `mappend'` can be derived from `mappend` by inlining these definitions and applying short cut fusion in conjunction with standard program simplification rules.

```
map :: forall a. forall b. (a -> b) -> List a -> List b
map = /\a b. \f l. build b (/\a'. \(n::a') (c :: b -> a' -> a').
                    foldr a a' n (\(y::a) (l'::a'). c (f y) l') l)

append :: forall a. List a -> List a -> List a
append = /\a. \xs ys. build a (/\b. \(n::b) (c::a -> b -> b).
                        foldr a b (foldr a b n c ys) c xs)
```

Fig. 2. Functions in build-foldr form.

1.2 The problem of correctness

Short cut fusion has successfully been used to improve programs in modern functional languages. It has even been shown to transform modular programs into monolithic ones exhibiting order-of-magnitude efficiency increases over those from which they are derived. Nevertheless, there remain difficulties associated with the use of short cut fusion. One of the most substantial is that its correctness has not yet been proved for the languages to which it is applied.

Short cut fusion has traditionally been treated purely syntactically, with little consideration given to the underlying semantics of the programs to which it is applied. In particular, the fact that the `foldr-build` rule holds only for languages admitting parametric models has been downplayed in the literature, and the application of short cut fusion to functional programs has been justified by appealing either to intuition about the operational behavior of `build` and `foldr` or to Wadler's "free theorems" (Wadler, 1989).[1] However, intuition is unsuitable as a basis for formal proofs, and the correctness of "free theorems" itself relies on the existence of parametric models. Since no parametric models for modern functional languages are known to exist, these justifications of short cut fusion for them are far from satisfactory.

Simply put, parametricity is the requirement that all polymorphic functions definable in a language operate uniformly over all types. This requirement gives rise to corresponding uniformity conditions on models, and these conditions are known to be satisfied by models supporting a parametric structure. Parametric models have been shown to exist for some higher-order polymorphic languages (Bainbridge *et al.*, 1990), but because these fail to model fixpoint recursion they do not adequately accommodate short cut fusion. While it may be possible to extend such models to encompass fixpoint recursion, this has not been reported in the literature, and until recently the existence of parametric models for languages supporting both higher-order polymorphic functions and fixpoint recursion had not been demonstrated. As a result, neither short cut fusion for even the most streamlined of higher-order polymorphic languages with fixpoint recursion, nor short cut fusion for the modern functional languages which extend them, has enjoyed a formal proof of correctness.

[1] In fact, the only formal proof of correctness of short cut fusion for a modern functional language on record appeals to Wadler's "free theorems" (Gill, 1996).

1.3 Proving correctness

In this paper we provide the first-ever formal proof of the correctness of short cut fusion for a calculus supporting both higher-order polymorphic functions and fixpoint recursion. Because modern functional languages typically support features that cannot be modeled in such calculi, our results do not apply to them directly. Nevertheless, our results do make some progress toward proving the correctness of short cut fusion for modern functional languages, and thus toward bridging the gap between the theory of parametricity and the practice of program fusion.

Our proof of the correctness of short cut fusion relies on Pitts' recent demonstration of the existence of relationally parametric models for a class of polymorphic lambda calculi supporting fixpoint recursion at the level of terms and recursion via data types with non-strict constructors at the level of types (Pitts, 2000, 1998b). Pitts uses logical relations to characterize contextual equivalence in such calculi, and this characterization enables him to show that identifying contextually equivalent terms gives rise to relationally parametric models for them. Our main result (Theorem 2.1) employs Pitts' characterization of contextual equivalence to demonstrate that programs in these calculi which have undergone short cut fusion are contextually equivalent to their unfused counterparts. The correctness of short cut fusion for them follows immediately.

Our proof techniques, like those of Pitts on which they are based, are operational in nature. Denotational approaches to proving the correctness of short cut fusion have thus far been unsuccessful. While it may be possible to construct a proof directly using the denotational notions that Pitts captures syntactically, to our knowledge this has not yet been accomplished. Similar remarks apply to directly constructing relationally parametric models of rank-2 fragments of suitable polymorphic calculi. It is worth noting that Pitts' relationally parametric characterization of contextual equivalence holds even in the presence of fully impredicative polymorphism. Characterization of contextual equivalence for predicative calculi (i.e. for calculi whose types do not rely on the collection of types for their definition) can be achieved by appropriately restricting the characterizations for the corresponding impredicative ones.

Short cut fusion can be generalized along two orthogonal dimensions. On the one hand, short cut fusion is easily generalized to arbitrary covariant recursive types – called *algebraic data types* below and elsewhere. Such generalizations have already been incorporated into a number of automatic fusion tools (Chitil, 1999; Gill, 1996; Johann, 1997; Johann & Visser, 2000; Németh, 2000). On the other hand, short cut fusion for lists can be generalized to accommodate more general list production. Gill (1996) has introduced a program construct called `augment` which generalizes `build` to produce lists with tails other than `Nil`. The behavior of `augment` is similar to that of `build`, the main difference being that whereas `build` instantiates the occurrence of the "abstract" list constructor corresponding to `Nil` in the result list specified by a list template `M` with `Nil` itself, `augment` can instantiate it with any given list. More specifically, if `t` is a type, `M` is any term with type `forall a. a -> (t -> a -> a) -> a`, and `ys` has type `List t`, then

```
augment t M ys = M (List t) ys Cons
```

Gill has also derived a `foldr-augment` fusion rule, similar to the `foldr-build` rule, for lists. According to this rule, occurrences of `foldr t t' n c (augment t M ys)` in a program can be replaced by `M t' (foldr t t' n c ys) c`.

In fact, short cut fusion can be generalized along both of these dimensions simultaneously to arrive at generalizations of `augment` and the `foldr-augment` rule for lists to non-list algebraic data types. In these more general settings, `augment` can be interpreted as constructing substitution instances of algebraic data structures, and the generalized `foldr-augment` rule can be viewed as optimizing compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them. The techniques in this paper can be extended in a straightforward – but notationally intensive – manner along the lines of Pitts (1998b) to prove the correctness of the generalized `foldr-augment` rule. Correctness of the `foldr-augment` rule for lists, as well as of short cut fusion for non-list algebraic data types, are immediate consequences. Details appear in Johann (2001).

2 PolyPCF and contextual equivalence

In exploring the correctness of short cut fusion we work in the same setting as in Pitts (2000), and our presentation is heavily influenced by that paper. In this section we introduce Pitts' PolyPCF, the polymorphic lambda calculus for which we formalize and prove the correctness of short cut fusion. We also make precise the notion of contextual equivalence which is required in this endeavor. Contextual equivalence for PolyPCF terms is characterized in section 3, which culminates in section 3.3 in a proof of correctness of short cut fusion for PolyPCF. Section 4 concludes.

2.1 PolyPCF

PolyPCF combines the Girard-Reynolds polymorphic lambda calculus with Plotkin's PCF by extending PCF with lazy lists and ∀-types. Since the treatment of ground types (e.g. natural numbers and booleans) in the theory developed here is precisely the same as the treatment of list types, for notational convenience we assume that PolyPCF supports only the latter. The syntax of PolyPCF types and terms is given in figure 3. As described in the introduction, the theory developed here extends to accommodate non-list algebraic data types.

A number of remarks concerning the definitions of figure 3 are in order. Type variables and term variables range over disjoint countably infinite sets. The constructions $\forall\alpha.(\_)$, $\lambda x : \tau.\_$, $\Lambda\alpha.\_$, and $\texttt{case}\ M\ \texttt{of}\ \{\texttt{Nil}_\tau \Rightarrow M \mid \texttt{Cons}_\tau\ x\ x' \Rightarrow \_\}$ are binders. As is customary, we identify types and terms which differ only by renamings of their bound variables. We write $ftv(e)$ for the (finite) set of free type variables of a type or term e, and $fv(M)$ for the (finite) set of free variables of a term M. The result of substituting the type τ for all free occurrences of the type variable α in a type or term e is denoted $e[\tau/\alpha]$. The result of substituting the term M' for all free occurrences of the variable x in the term M is denoted $M[M'/x]$.

Types	τ	$::=$	α	type variable
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\forall\alpha.\tau$	$\forall$-type
		$\mid$	$List\ \tau$	list type
Terms	M	$::=$	x	variable
		$\mid$	$\lambda x : \tau.\, M$	function abstraction
		$\mid$	$M M$	function application
		$\mid$	$\Lambda\alpha.\, M$	type abstraction
		$\mid$	$M\tau$	type application
		$\mid$	$\mathtt{fix}(M)$	fixpoint recursion
		$\mid$	$\mathtt{Nil}_\tau$	empty list
		$\mid$	$\mathtt{Cons}_\tau\ M\ M$	non-empty list
		$\mid$	$\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{Nil}_\tau \Rightarrow M \mid \mathtt{Cons}_\tau\ x\ x \Rightarrow M\}$	case expression

Fig. 3. Syntax of PolyPCF.

$$\Gamma, x : \tau \vdash x : \tau \qquad\qquad \frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \mathtt{fix}(F) : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\, M : \tau_1 \to \tau_2} \qquad\qquad \frac{\Gamma \vdash F : \tau_1 \to \tau_2 \qquad \Gamma \vdash A : \tau_1}{\Gamma \vdash F\,A : \tau_2}$$

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau} \qquad\qquad \frac{\Gamma \vdash G : \forall\alpha.\tau_1}{\Gamma \vdash G\,\tau_2 : \tau_1[\tau_2/\alpha]}$$

$$\Gamma \vdash \mathtt{Nil}_\tau : List\ \tau \qquad\qquad \frac{\Gamma \vdash H : \tau \qquad \Gamma \vdash T : List\ \tau}{\Gamma \vdash \mathtt{Cons}_\tau\ H\ T : List\ \tau}$$

$$\frac{\Gamma \vdash L : List\ \tau \qquad \Gamma \vdash M_1 : \tau_2 \qquad \Gamma, h : \tau_1, t : List\ \tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \mathtt{case}\ L\ \mathtt{of}\ \{\mathtt{Nil}_\tau \Rightarrow M_1 \mid \mathtt{Cons}_\tau\ z\ zs \Rightarrow M_2\} : \tau_2}$$

Fig. 4. PolyPCF type assignment.

We will be concerned only with PolyPCF terms which are typeable. The type assignment relation for PolyPCF is completely standard; it is given in figure 4. A *typing environment* Γ is a pair A, Δ, where A is a finite set of type variables and Δ is a function defined on a finite set $dom(\Delta)$ of variables which maps each $x \in dom(\Delta)$ to a type with free type variables in A. We write $\Gamma \vdash M : \tau$ to indicate that term M has type τ in the type environment Γ. Implicit in this notation are four assumptions, namely that $\Gamma = A, \Delta$, that $ftv(M) \subseteq A$, that $ftv(\tau) \subseteq A$, and that $fv(M) \subseteq dom(\Delta)$. The notation $\Gamma, x : \tau$ indicates the typing environment obtained from $\Gamma = A, \Delta$ by extending the function Δ to map $x \notin dom(\Delta)$ to τ. Similarly, the notation Γ, α denotes the typing environment obtained from $\Gamma = A, \Delta$ by extending A with a type variable $\alpha \notin A$.

The explicit type annotations on lambda-bound term variables and empty lists ensure that well-formed PolyPCF terms have unique types. That is, given Γ and

$$V \Downarrow V \;(V \text{ a value}) \qquad \frac{F\,\mathtt{fix}(F) \Downarrow V}{\mathtt{fix}(F) \Downarrow V}$$

$$\frac{L \Downarrow \mathtt{Cons}\; H\; T \qquad M_2[H/z, T/zs] \Downarrow V}{\mathtt{case}\; L\; \mathtt{of}\; \{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\; z\; zs \Rightarrow M_2\} \Downarrow V} \qquad \frac{F \Downarrow \lambda x:\tau.\, M \qquad M[A/x] \Downarrow V}{F\,A \Downarrow V}$$

$$\frac{L \Downarrow \mathtt{Nil} \qquad M_1 \Downarrow V}{\mathtt{case}\; L\; \mathtt{of}\; \{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\; z\; zs \Rightarrow M_2\} \Downarrow V} \qquad \frac{G \Downarrow \Lambda\alpha.\, M \qquad M[\tau/\alpha] \Downarrow V}{G\,\tau \Downarrow V}$$

Fig. 5. PolyPCF evaluation relation.

M, there is at most one type τ for which $\Gamma \vdash M : \tau$ holds. For convenience we sometimes suppress type information below.

A type τ is *closed* if $ftv(\tau) = \emptyset$. A term M is *closed* if $fv(M) = \emptyset$, regardless of whether or not M contains free type variables. The set of closed PolyPCF terms is denoted *Typ*. For $\tau \in Typ$ the set of closed PolyPCF terms M for which $\emptyset, \emptyset \vdash M : \tau$ is denoted $Term(\tau)$. Both `foldr` and `build` are expressible as closed PolyPCF terms: writing l_τ for the type $\forall\beta.\, \beta \to (\tau \to \beta \to \beta) \to \beta$ we can define

$$\mathtt{foldr} \;=\; \Lambda\alpha.\, \Lambda\beta.\, \lambda n : \beta.\, \lambda c : \alpha \to \beta \to \beta.\, \lambda xs : List\;\alpha.\, \mathtt{unbuild}\; \alpha\; xs\; \beta\; n\; c$$

and

$$\mathtt{build} = \Lambda\alpha.\, \mathtt{construct}\; \alpha$$

where

$$\begin{aligned}\mathtt{unbuild}\;\tau \;&=\; \mathtt{fix}(\lambda h : List\;\tau \to l_\tau.\lambda xs : List\;\tau.\, \Lambda\alpha.\, \lambda n : \alpha.\lambda c : \tau \to \alpha \to \alpha.\\ &\qquad\qquad \mathtt{case}\; xs\; \mathtt{of}\; \{\mathtt{Nil}_\tau \Rightarrow n \mid \mathtt{Cons}_\tau\; z\; zs \Rightarrow c\, z\, (h\; zs\; \alpha\, n\; c)\})\end{aligned}$$

and

$$\mathtt{construct}\;\tau \;=\; \lambda M : l_\tau.\, M\; (List\;\tau)\; \mathtt{Nil}_\tau\; (\lambda h : \tau.\, \lambda t : List\;\tau.\, \mathtt{Cons}_\tau\; h\; t)$$

The auxiliary terms `unbuild` and `construct` are necessary because the type constructor *List* must be applied to a type to produce a well-formed PolyPCF type. That is, *List* itself is not a PolyPCF type. In addition, the definition of `construct` reflects the fact that the list constructors `Nil` and `Cons` must be fully applied in well-formed PolyPCF terms.

2.2 *Operational semantics*

The operational semantics of PolyPCF is given by the inductively defined evaluation relation in figure 5. It relates a closed term M to a value V of the same closed type; this is denoted $M \Downarrow V$. The set of PolyPCF *values* is given by

$$V ::= \lambda x.\, M \;\mid\; \Lambda\alpha.\, M \;\mid\; \mathtt{Nil} \;\mid\; \mathtt{Cons}\; M\; M$$

Note that function application is given a call-by-name semantics, constructors are non-strict, and type applications are not evaluated "under the Λ." In addition, PolyPCF evaluation is deterministic, although the rule for `fix` entails the existence of terms whose evaluation does not terminate.

2.3 Contextual equivalence

With the operational semantics of PolyPCF in place, we can now make precise the notion of contextual equivalence for its terms. Informally, two terms in a programming language are contextually equivalent if they are interchangeable in any program with no difference in observable behavior when the resulting programs are executed. To formalize this notion for PolyPCF we must specify what a PolyPCF program is, as well as the PolyPCF program behavior we are interested in observing.

Recall that ground types have been replaced by list types in PolyPCF. To mimic the standard notions of a program as a closed term of ground type and the observable behavior of a program as the constant value, if any, to which it evaluates, we therefore define a PolyPCF *program* to be a closed term of list type and take the *observable behavior* of a PolyPCF program to be whether or not it evaluates to `Nil`.[2] We further define two PolyPCF terms M_1 and M_2 such that $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$ to be *contextually equivalent with respect to* Γ if, for any context $\mathscr{M}[\_]$ for which $\mathscr{M}[M_1], \mathscr{M}[M_2] \in \mathit{Term}(\mathit{List}\ \tau')$ for $\tau' \in \mathit{Typ}$, we have

$$\mathscr{M}[M_1] \Downarrow \mathtt{Nil}_{\tau'} \Leftrightarrow \mathscr{M}[M_2] \Downarrow \mathtt{Nil}_{\tau'}$$

As usual, a *context* $\mathscr{M}[\_]$ is a PolyPCF term with a subterm replaced by the placeholder '_', and $\mathscr{M}[M]$ denotes the term which results from replacing the placeholder by the term M. We write

$$\Gamma \vdash M_1 =_{ctx} M_2 : \tau$$

to indicate that M_1 and M_2 are terms of type τ which are contextually equivalent with respect to Γ. If M_1 and M_2 are closed terms of closed type τ, we write $M_1 =_{ctx} M_2 : \tau$ instead of $\emptyset, \emptyset \vdash M_1 =_{ctx} M_2 : \tau$. In this case we say simply that M_1 and M_2 are *contextually equivalent*.

For terms M, M_1, and M_2 of type τ_1, A of type τ_2, and F of type τ, the following contextual equivalences are shown in Pitts (2000) to hold:

$$(\lambda x : \tau_2 . M)A =_{ctx} M[A/x] : \tau_1 \tag{1}$$

$$(\Lambda\alpha . M)\tau_2 =_{ctx} M[\tau_2/\alpha] : \tau_1[\tau_2/\alpha] \tag{2}$$

$$\mathtt{case}\ \mathtt{Nil}_{\tau_2}\ \mathtt{of}\ \{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\ z\ zs \Rightarrow M_2\} =_{ctx} M_1 : \tau_1 \tag{3}$$

$$\begin{aligned}&\mathtt{case}\ \mathtt{Cons}_{\tau_2}\ H\ T\ \mathtt{of}\\ &\quad\{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\ z\ zs \Rightarrow M_2\} =_{ctx} M_2[H/z, T/zs] : \tau_1\end{aligned} \tag{4}$$

$$\mathtt{fix}(F) =_{ctx} F\,\mathtt{fix}(F) : \tau \tag{5}$$

[2] It may seem more natural to observe as much as one can about the results of evaluation. But observing the entire list value, if any, to which a program evaluates – rather than just observing whether or not it evaluates to `Nil` – can entail the comparison of thunks, and this leads to too high a degree of intensionality. On the other hand, by considering programs in suitable contexts, we can show that observing whether or not they evaluate to `Nil` leads to the same notion of observational equivalence as observing the outermost constructors of the list values, if any, to which programs evaluate. The same technique can also be used to show that merely observing whether or not programs terminate leads once again to this same notion of observational equivalence. These alternative characterizations may seem more intuitive.

2.4 Formalizing short cut fusion

Once we have the notion of contextual equivalence at our disposal we can formalize the correctness of short cut fusion for PolyPCF. We will consider only closed types and terms below. This restriction is reasonable because contextual equivalence for open terms is reducible to contextual equivalence for closed terms of closed type, as shown in Theorem 5.1 of Pitts (2000).

Theorem 2.1
(Short Cut Fusion) Let τ and τ' be closed types, and let

$$M : \forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha,$$

$$n : \tau',$$

and

$$c : \tau \to \tau' \to \tau'$$

be closed terms. Then

$$\mathtt{foldr}\;\tau\;\tau'\;n\;c\;(\mathtt{build}\;\tau\;M) =_{ctx} M\;\tau'\;n\;c \;\;:\;\; \tau'$$

3 Correctness of short cut fusion

To prove the correctness of short cut fusion for PolyPCF we would like to define a logical relation which coincides with PolyPCF contextual equivalence. A logical relation R is a collection $\{R^\tau \mid \tau \text{ a type}\}$ of relations with the property that the relations at complex types are determined by the relations at their subtypes in such a way that closure of R under the basic operations of term formation is guaranteed. A logical relation which coincides with PolyPCF contextual equivalence would enforce contextual equivalence of related terms. This would in turn incorporate into the theory of contextual equivalence a notion of relational parametricity analogous to that introduced by Reynolds for the pure polymorphic lambda calculus (Reynolds, 1983).

Unfortunately, a naive approach to defining a logical relation with the desired properties (i.e. an approach which quantifies over *all* appropriately typed relations in defining the relation at $\forall$-types) is not sufficiently restrictive to ensure parametricity. What is needed is some criterion for identifying those relations on closed PolyPCF terms which are “admissible for fixpoint induction,” in the sense that they syntactically capture the domain-theoretic notion of admissibility. (In domain theory, a subset of a domain is said to be *admissible* if it contains the least element of the domain and is closed under taking least upper bounds of chains in the domain.) The notion of $\top\top$-closure defined below, taken from Pitts (2000), provides a criterion sufficient to guarantee this kind of admissibility (Abadi, 2000).

The notion of $\top\top$-closure is induced by a Galois connection between term relations and evaluation contexts, i.e. contexts $\mathscr{M}[\_]$ which have a single occurrence of the placeholder ‘_’ in the position at which the next subexpression will be evaluated. In Pitts (2000), analysis of evaluation contexts is aided by recasting them

$$\Gamma \vdash \mathit{Id} : \tau \hookrightarrow \tau$$

$$\frac{\Gamma \vdash S : \tau' \hookrightarrow \tau'' \qquad \Gamma \vdash M : \tau}{\Gamma \vdash S \circ (\_\,M) : (\tau \hookrightarrow \tau') \hookrightarrow \tau''} \qquad \frac{\Gamma \vdash S : \tau'[\tau/\alpha] \hookrightarrow \tau'' \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash S \circ (\_\,\tau) : (\forall \alpha.\tau') \hookrightarrow \tau''}$$

$$\frac{\Gamma \vdash S : \tau' \hookrightarrow \tau'' \qquad \Gamma \vdash M_1 : \tau' \qquad \Gamma, h : \tau, t : \mathit{List}\,\tau \vdash M_2 : \tau'}{\Gamma \vdash S \circ (\texttt{case}\,\_\,\texttt{of}\,\{\texttt{Nil} \Rightarrow M_1 \mid \texttt{Cons}\,h\,t \Rightarrow M_2\}) : \mathit{List}\,\tau \hookrightarrow \tau''}$$

Fig. 6. Frame stack type judgements.

in terms of the notion of frame stack given in Definition 3.1 below. This frame stack realization of evaluation contexts gives rise to Pitts' syntactic characterization of the PolyPCF termination properties entailed by contextual equivalence. The resulting PolyPCF *structural termination relation* provides the key to appropriate specification of the clause for $\forall$-types in the logical relation which coincides with contextual equivalence.

After sketching Pitts' characterization of contextual equivalence in terms of logical relations in sections 3.1 and 3.2, we use it in section 3.3 to prove correctness of short cut fusion. This proof is the main contribution of the paper.

3.1 $\top\top$-*closed relations*

Definition 3.1
The grammar for PolyPCF *frame stacks* is

$$S ::= \mathit{Id} \mid S \circ F$$

where F ranges over *frames*:

$$F ::= (\_\,M) \mid (\_\,\tau) \mid \texttt{case}\,\_\,\texttt{of}\,\{\ldots\}$$

Frame stacks have types and typing derivations, although explicit type information is not included in their syntax. The type judgement $\Gamma \vdash S : \tau \hookrightarrow \tau'$ for a frame stack S indicates the argument type τ and the result type τ' of S. As usual, Γ is a typing environment and certain well-formedness conditions of judgements hold; in particular, Γ is assumed to contain all free variables and free type variables of all expressions appearing in the judgement. The axioms and rules inductively defining type judgements for frame stacks are given in figure 6. We will only be concerned with stacks which are typeable. Although well-formed frame stacks do not have unique types, they do satisfy the following property: Given Γ, S, and τ, there is at most one τ' such that $\Gamma \vdash S : \tau \hookrightarrow \tau'$ holds. In this paper, the argument types of frame stacks will always be known at the time of their use.

Given closed types τ and τ', we write $\mathit{Stack}(\tau, \tau')$ for the set of frame stacks for which $\emptyset, \emptyset \vdash S : \tau \hookrightarrow \tau'$. We are particularly interested in the case when the result type τ' of a frame stack is a list type, and so we write

$$\mathit{Stack}(\tau) = \bigcup\{\mathit{Stack}(\tau, \mathit{List}\,\tau') \mid \tau' \in \mathit{Typ}\}$$

$$\frac{S \top M[A/x]}{S \circ (\_\, A) \top \lambda x : \tau.\, M} \qquad \frac{S \circ (\_\, A) \top F}{S \top F\, A}$$

$$\frac{S \top M[\tau/\alpha]}{S \circ (\_\, \tau) \top \Lambda\alpha.M} \qquad \frac{S \circ (\_\, \tau) \top G}{S \top G\tau}$$

$$\frac{S \circ (\_\, \mathtt{fix}(F)) \top F}{S \top \mathtt{fix}(F)} \qquad \frac{}{Id \top \mathtt{Nil}}$$

$$\frac{S \top M_1}{S \circ (\mathtt{case}\ \_\ \mathtt{of}\ \{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\ z\ zs \Rightarrow M_2\}) \top \mathtt{Nil}}$$

$$\frac{S \top M_2[H/z, T/zs]}{S \circ (\mathtt{case}\ \_\ \mathtt{of}\ \{\mathtt{Nil} \Rightarrow M_1 \mid \mathtt{Cons}\ z\ zs \Rightarrow M_2\}) \top \mathtt{Cons}\ H\ T}$$

Fig. 7. PolyPCF structural termination relation.

The operation $S, M \mapsto S M$ of *applying a stack to a term* is the analogue for frame stacks of the operation of filling the hole in an evaluation context with a term. It is defined by induction on the number of frames in the stack as follows:

$$\begin{array}{lcl} Id\ M & = & M \\ (S \circ F)\ M & = & S(F[M]) \end{array}$$

Here, $F[M]$ is the term that results from replacing '_' by M in the frame F. If $S \in Stack(\tau, \tau')$ and $M \in Term(\tau)$, then $S M \in Term(\tau')$. Unlike PolyPCF evaluation, stack application is strict in its second argument. This follows from the fact that

$$S\ M \Downarrow V \text{ iff there exists a value } V' \text{ such that } M \Downarrow V' \text{ and } S\ V' \Downarrow V$$

which can be proved by induction on the number of frames in the frame stack S. The corresponding property

$$F[M] \Downarrow V \text{ iff there exists a value } V' \text{ such that } M \Downarrow V' \text{ and } F[V'] \Downarrow V$$

for frames, needed for the base case of the induction, follows directly from the inductive definition of the PolyPCF evaluation relation in figure 5.

PolyPCF termination is captured by the *structural termination relation* $(\_)\top(\_)$ defined in figure 7. For all closed types τ and τ', all frame stacks $S \in Stack(\tau, List\ \tau')$, and all $M \in Term(\tau)$, we have

$$S\ M \Downarrow \mathtt{Nil}_{\tau'} \Leftrightarrow S \top M$$

Pitts uses this characterization of PolyPCF termination to prove that, in any context, evaluation of a fixed point terminates iff some finite unwinding of it does. This, in turn, allows him to make precise the sense in which $\top\top$-closed relations – defined below – are admissible for fixed point induction.

Definition 3.2
A PolyPCF *term relation* is a binary relation between (typeable) closed terms. Given closed types τ and τ' we write $Rel(\tau, \tau')$ for the set of term relations which are

subsets of $Term(\tau) \times Term(\tau')$. A PolyPCF *stack relation* is a binary relation between (typeable) frame stacks whose result types are list types. We write $Rel^{\top}(\tau, \tau')$ for the set of relations which are subsets of $Stack(\tau) \times Stack(\tau')$.

The relation $(\_)^{\top}$ transforms stack relations into term relations, and vice versa:

Definition 3.3
Given any closed types τ and τ', and any $r \in Rel(\tau, \tau')$, define $r^{\top} \in Rel^{\top}(\tau, \tau')$ by

$$(S, S') \in r^{\top} \;\Leftrightarrow\; \text{for all } (M, M') \in r.\, S \top M \Leftrightarrow S' \top M'$$

Similarly, given any $s \in Rel^{\top}(\tau, \tau')$, define $s^{\top} \in Rel(\tau, \tau')$ by

$$(M, M') \in s^{\top} \;\Leftrightarrow\; \text{for all } (S, S') \in s.\, S \top M \Leftrightarrow S' \top M'$$

The relation $(\_)^{\top}$ gives rise to the notion of $\top\top$-closure which characterizes those relations which are suitable for consideration in the clause for $\forall$-types in the definition of the logical relation which coincides with contextual equivalence.

Definition 3.4
A term relation r is said to be $\top\top$-*closed* if $r = r^{\top\top}$.

Since $r \subseteq r^{\top\top}$ always holds, this is equivalent to requiring that $r^{\top\top} \subseteq r$. Expanding the definitions of $r^{\top}$ and $s^{\top}$ above gives $(M, M') \in r^{\top\top}$ iff

$$\begin{array}{c} \text{for each pair } (S, S') \text{ of (appropriately typed) stacks,} \\ \text{if for all } (N, N') \in r.\, S \top N \;\Leftrightarrow\; S' \top N', \\ \text{then } S \top M \;\Leftrightarrow S' \top M'. \end{array} \tag{6}$$

This characterization of $\top\top$-closedness will be used in section 3.3.

3.2 Characterizing contextual equivalence

We are now in a position to describe PolyPCF contextual equivalence in terms of parametric logical relations. The following constructions on term relations describe the ways in which the various PolyPCF type constructors act on term relations.

Definition 3.5
Action of $\to$ on term relations: given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, define $r_1{\to}r_2 \in Rel(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$ by

$$(F, F') \in r_1{\to}r_2 \;\Leftrightarrow\; \text{for all } (A, A') \in r_1.\, (FA, F'A') \in r_2$$

Action of $\forall$ on term relations: let τ_1 and τ_1' be types with at most one free type variable, say α, and let R be a function mapping term relations $r \in Rel(\tau_2, \tau_2')$ for any closed types τ_2 and τ_2' to term relations $R(r) \in Rel(\tau_1[\tau_2/\alpha], \tau_1'[\tau_2'/\alpha])$. Define the term relation $\forall r.\, R(r) \in Rel(\forall\alpha.\tau_1, \forall\alpha.\tau_1')$ by

$$(G, G') \in \forall r.\, R(r) \;\Leftrightarrow\; \text{for all } \tau_2, \tau_2' \in Typ.\ \text{for all } r \in Rel(\tau_2, \tau_2').\ (G\tau_2, G'\tau_2') \in R(r)$$

Action of *List* $(\_)$ on term relations: let τ and τ' be closed types, let $r_1 \in Rel(\tau, \tau')$, and let $r_2 \in Rel(List\ \tau, List\ \tau')$. Define $1 + (r_1 \times r_2) \in Rel(List\ \tau, List\ \tau')$ by

$$\begin{array}{l} 1 + (r_1 \times r_2) = \\ \quad \{(\mathtt{Nil}_{\tau}, \mathtt{Nil}_{\tau'})\} \cup \{(\mathtt{Cons}\ H\ T, \mathtt{Cons}\ H'\ T') \mid (H, H') \in r_1 \text{ and } (T, T') \in r_2\} \end{array}$$

Since $Rel(List\ \tau, List\ \tau')$ is a complete lattice under the subset relation, and since, for each r_1, the mapping $r_2 \mapsto (1+(r_1 \times r_2))^{\top\top}$ is monotone, we can form its greatest fixed point. Denoting the greatest fixed point of a mapping $r \mapsto R(r)$ by $\nu r.R(r)$, for each relation r_1 we define the term relation $List\ r_1$ by

$$List\ r_1 = \nu r_2.\,(1+(r_1 \times r_2))^{\top\top}$$

We form the greatest fixed point here because contextual equivalence does not distinguish between programs unless there are observable reasons for doing so.

Using these notions of actions we can define the logical relations in which we are interested.

Definition 3.6
The logical relation Δ comprises a family of mappings

$$r_1 \in Rel(\tau_1, \tau_1'), \ldots, r_n \in Rel(\tau_n, \tau_n') \mapsto \Delta_\tau(\overline{r_n}/\overline{\alpha_n}) \in Rel(\tau[\overline{\tau_n}/\overline{\alpha_n}], \tau[\overline{\tau_n'}/\overline{\alpha_n}]) \qquad (7)$$

from tuples of term relations to term relations, one for each type τ and each list $\overline{\alpha_n}$ of distinct variables containing the free variables of τ. These mappings are defined by the four clauses given below.

1. $\Delta_\alpha(r/\alpha, \overline{r_n}/\overline{\alpha_n}) = r$
2. $\Delta_{\tau_1 \to \tau_2}(\overline{r_n}/\overline{\alpha_n}) = \Delta_{\tau_1}(\overline{r_n}/\overline{\alpha_n}) \to \Delta_{\tau_2}(\overline{r_n}/\overline{\alpha_n})$
3. $\Delta_{\forall\alpha.\tau}(\overline{r_n}/\overline{\alpha_n}) = \forall r.\,\Delta_\tau(r^{\top\top}/\alpha, \overline{r_n}/\overline{\alpha_n})$
4. $\Delta_{List\ \tau}(\overline{r_n}/\overline{\alpha_n}) = List\,(\Delta_\tau(\overline{r_n}/\overline{\alpha_n}))$

To see that the third clause in Definition 3.6 is sensible, note that $\tau[\overline{\tau_n}/\overline{\alpha_n}]$ and $\tau[\overline{\tau_n'}/\overline{\alpha_n}]$ are types containing at most one free variable, namely α, and that Δ_τ maps any term relation $r \in Rel(\sigma, \sigma')$ for closed types σ, σ' to the term relation $\Delta_\tau(r^{\top\top}/\alpha, \overline{r_n}/\overline{\alpha_n}) \in Rel(\tau[\overline{\tau_n}/\overline{\alpha_n}][\sigma/\alpha], \tau[\overline{\tau_n'}/\overline{\alpha_n}][\sigma'/\alpha])$. According to Definition 3.5, we therefore have $\forall r.\,\Delta_\tau(r^{\top\top}/\alpha, \overline{r_n}/\overline{\alpha_n}) \in Rel(\forall\alpha.\tau[\overline{\tau_n}/\overline{\alpha_n}], \forall\alpha.\tau[\overline{\tau_n'}/\overline{\alpha_n}])$, as required by (7).

Taking $n=0$ in (7), we see that for each closed type τ we can apply Δ_τ to the empty tuple of term relations to obtain the term relation $\Delta_\tau() \in Rel(\tau, \tau)$. It is shown in Pitts (2000) that this relation coincides with the relation of contextual equivalence of closed terms at the closed type τ. In fact, Pitts shows a stronger correspondence between Δ and contextual equivalence: using an appropriate notion of closing substitution to extend Δ to a logical relation $\Gamma \vdash M\,\Delta\,M' : \tau$ between open terms, he shows that

$$\Gamma \vdash M =_{ctx} M' : \tau \quad \Leftrightarrow \quad \Gamma \vdash M\,\Delta\,M' : \tau \qquad (8)$$

Observation (8) guarantees that the logical relation Δ corresponds to the operational semantics of PolyPCF. In particular, the definition of $\Delta_{\tau_1 \to \tau_2}$ in the second clause of Definition 3.6 reflects the fact that termination at function types is not observable in PolyPCF. This is as expected: for types τ_1 and τ_2, the relation $\Delta_{\tau_1}(\overline{r_n}/\overline{\alpha_n}) \to \Delta_{\tau_2}(\overline{r_n}/\overline{\alpha_n})$ may not be $\top\top$-closed, and so may not capture PolyPCF contextual equivalence.

It is possible to define call-by-value and lazy versions of PolyPCF. As pointed out in Pitts (2000), both versions require modification of the definition of the relation

$(\_)\top(\_)$, as well as modification of the action of arrow types on term relations to reflect the appropriate operational semantics and notions of observability. In addition, defining a call-by-value PolyPCF also requires a slightly different notion of frame stack. The full development of these ideas for a call-by-value version of PolyPCF is given in Pitts (1998a). The details for a lazy PolyPCF remain unpublished. Laziness is necessary, for example, to capture the semantics of languages such as Haskell, whose termination at function types is observable. (Existence of the function `seq` guarantees that termination at function types is observable in Haskell. This function takes two arguments and reduces the first to weak head normal form before returning the second.)

For our purposes we need only the following two corollaries of (8). Proposition 3.7 guarantees that Δ is reflexive.

Proposition 3.7
For each closed type τ and each closed term M, $(M, M) \in \Delta_\tau()$.

Proposition 3.8
For all closed types τ and closed terms M and M' of type τ,

$$M =_{ctx} M' : \tau \quad \Leftrightarrow \quad \text{for all } S \in \mathit{Stack}(\tau).\ S \top M \ \Leftrightarrow\ S \top M'$$

3.3 *Proof of Theorem 2.1*

Let Δ be as in Definition 3.6, and let τ, τ', M, n, and c be as in the statement of Theorem 2.1. Since M and its type are closed, Proposition 3.7 ensures that

$$(M, M) \in \Delta_{\forall\alpha.\,\alpha\to(\tau\to\alpha\to\alpha)\to\alpha}() \tag{9}$$

Applying the definition of Δ for $\forall$-types shows that (9) holds iff for all closed types τ'' and τ' and for all $r \in \mathit{Rel}(\tau'', \tau')$,

$$(M\tau'', M\tau') \in \Delta_{\alpha\to(\tau\to\alpha\to\alpha)\to\alpha}(r^{\top\top}/\alpha)$$

Two-fold application of the definition of Δ for arrow types ensures that for all closed types τ'' and τ', and for all $r \in \mathit{Rel}(\tau'', \tau')$, for all pairs of closed terms $(n', n) \in r^{\top\top}$ and $(c', c) \in \Delta_{\tau\to\alpha\to\alpha}(r^{\top\top}/\alpha)$, (9) holds iff

$$(M\ \tau''\ c'\ n', M\ \tau'\ c\ n) \in r^{\top\top}$$

Expanding the condition on (c', c) shows it equivalent to the assertion that if $(a', a) \in \Delta_\tau(r^{\top\top}/\alpha)$ and $(b', b) \in r^{\top\top}$, then $(c'\ a'\ b', c\ a\ b) \in r^{\top\top}$. Since (9) holds, we conclude that for all closed types τ'' and τ' and for all $r \in \mathit{Rel}(\tau'', \tau')$,

$$\begin{aligned} &\text{if } (n', n) \in r^{\top\top}, \\ &\qquad \text{and if } (a', a) \in \Delta_\tau(r^{\top\top}/\alpha) \text{ and } (b', b) \in r^{\top\top} \text{ imply } (c'\ a'\ b', c\ a\ b) \in r^{\top\top}, \\ &\qquad\quad \text{then } (M\ \tau''\ c'\ n', M\ \tau'\ c\ n) \in r^{\top\top} \end{aligned} \tag{10}$$

Note that all of the terms appearing in (10) are closed.

Now consider the instantiation

$$\begin{aligned}
\tau'' &= \mathit{List}\ \tau \\
r &= \{(M, M') \mid \texttt{foldr}\,\tau\,\tau'\ n\ c\ M\ =_{ctx}\ M'\ :\ \tau'\} \\
c' &= \lambda x.\,\lambda y.\,\texttt{Cons}\ x\ y \\
n' &= \texttt{Nil}
\end{aligned}$$

If we can verify that the hypotheses of (10) hold, then we may conclude that

$$\texttt{foldr}\,\tau\,\tau'\ n\ c\ (M\,(\mathit{List}\ \tau)\ \texttt{Nil}\ (\lambda x.\,\lambda y.\texttt{Cons}\ x\ y))\ =_{ctx}\ M\,\tau'\ n\ c\ :\ \tau'$$

Then, since $\texttt{build}\ \tau\ M =_{ctx} M\ (\mathit{List}\ \tau)\ \texttt{Nil}\ (\lambda x.\lambda y.\texttt{Cons}\ x\ y) : \mathit{List}\ \tau$, we will have proved the correctness of short cut fusion.

To verify that (10) holds we first prove that r is $\top\top$-closed. To see this, suppose $(M, M') \in r^{\top\top}$. We want to verify that $\texttt{foldr}\,\tau\,\tau'\ n\ c\ M =_{ctx} M'\ :\ \tau'$. Let $S \in \mathit{Rel}(\tau, \tau')$ be the "stack equivalent"

$$\begin{aligned}
&\mathit{Id} \circ \texttt{case}\ \_\ \texttt{of} \\
&\qquad \texttt{Nil} \Rightarrow n \\
&\qquad \texttt{Cons}\ z\ zs \Rightarrow c\ z\ (\texttt{foldr}\ \tau\ \tau'\ n\ c\ zs)
\end{aligned}$$

of the evaluation context $\texttt{foldr}\,\tau\,\tau'\ n\ c$. Then S is such that for all $N : \mathit{List}\ \tau$,

$$S\,N\ =_{ctx}\ \texttt{foldr}\,\tau\,\tau'\ n\ c\ N\ :\ \tau' \tag{11}$$

since

$$\begin{aligned}
\texttt{foldr}\,\tau\,\tau'\ n\ c\ N\ =_{ctx}\ & (\Lambda\alpha.\,\Lambda\beta.\,\lambda n.\,\lambda c.\,\lambda xs. \\
&\qquad \texttt{case}\ xs\ \texttt{of} \\
&\qquad\quad \{\texttt{Nil} \Rightarrow n\mid \\
&\qquad\quad\ \texttt{Cons}\ z\ zs \Rightarrow c\ z\ (\texttt{foldr}\,\alpha\,\beta\ n\ c\ zs)\})\ \tau\ \tau'\ n\ c\ N \\
=_{ctx}\ & \texttt{case}\ N\ \texttt{of} \\
&\quad \{\texttt{Nil} \Rightarrow n\mid \\
&\quad\ \texttt{Cons}\ z\ zs \Rightarrow c\ z\ (\texttt{foldr}\,\tau\,\tau'\ n\ c\ zs)\} \\
=_{ctx}\ & (\mathit{Id} \circ \texttt{case}\ \_\ \texttt{of} \\
&\qquad \{\texttt{Nil} \Rightarrow n\mid \\
&\qquad\ \texttt{Cons}\ z\ zs \Rightarrow c\ z\ (\texttt{foldr}\,\tau\,\tau'\ n\ c\ zs)\})\ N \\
=_{ctx}\ & S\ N
\end{aligned}$$

The first equivalence is by (5) and the definition of `foldr`, the second is by repeated application of (1) and (2), the third is by the definition of frame stack application, and the fourth is by the definition of S.

Observe that if we define the append operation on frame stacks by

$$S @ \mathit{Id} = S$$

and

$$S' @ (S \circ F) = (S' @ S) \circ F$$

then

$$(S' @ S) \top M\ \Leftrightarrow\ S' \top (S\ M) \tag{12}$$

Moreover, for any $S' \in \mathit{Stack}(\tau')$, the frame stack $(S' @ S,\ S')$ has the property that for all (N, N') with $\texttt{foldr}\, \tau\, \tau'\, n\, c\, N \;=_{ctx}\; N' \;:\; \tau'$,

$$(S' @ S) \top N \;\Leftrightarrow\; S' \top S\, N \;\Leftrightarrow\; S' \top N'$$

The first equivalence by (12), and the second is by Proposition 3.8 and (11) and the fact that $=_{ctx}$ is transitive. Together with (6), the fact that $(M, M') \in r^{\top\top}$ therefore implies that

$$(S' @ S) \top M \;\Leftrightarrow\; S' \top M' \tag{13}$$

But then

$$S' \top M' \;\Leftrightarrow\; (S' @ S) \top M \;\Leftrightarrow\; S' \top S\, M \;\Leftrightarrow\; S' \top \texttt{foldr}\, \tau\, \tau'\, n\, c\, M$$

Here, the first equivalence is by (13), the second is by (12), and the third is by (11). Since S' was arbitrary we have shown that

$$\text{for all } S' \in \mathit{Stack}(\tau').\;\; S' \top M' \;\Leftrightarrow\; S' \top \texttt{foldr}\, \tau\, \tau'\, n\, c\, M$$

By Proposition 3.8, we therefore have $M' =_{ctx} \texttt{foldr}\, \tau\, \tau'\, n\, c\, M \;:\; \tau'$, i.e., $(M, M') \in r$, as desired.

To verify the hypotheses of (10), first observe that $\texttt{foldr}\, \tau\, \tau'\, n\, c\, \texttt{Nil} =_{ctx} n : \tau'$, i.e., that $(n', n) \in r$. Second, note that since τ is closed, $\Delta_\tau(r^{\top\top}/\alpha)$ is precisely $\Delta_\tau()$. Thus, if $(a', a) \in \Delta_\tau(r^{\top\top}/\alpha)$, then by Proposition 3.7, then $a' =_{ctx} a : \tau$. If, in addition, $(b', b) \in r$, then $\texttt{foldr}\, \tau\, \tau'\, n\, c\, b' =_{ctx} b : \tau'$. Since $=_{ctx}$ is a congruence, equivalences (1) through (5) guarantee that

$$\texttt{foldr}\, \tau\, \tau'\, n\, c\, (c'\; a'\; b') \;=_{ctx}\; c\; a\; b \;:\; \tau'$$

It is also possible to derive $\top\top$-closedness of r as a consequence of Lemma 6.1 of Pitts (2000), but in the interest of keeping this paper as self-contained as possible, we choose to prove it directly.

4 Conclusion

In this paper, we have used Pitts' characterization of contextual equivalence for PolyPCF to provide the first proof of correctness of short cut fusion for a polymorphic lambda calculus supporting fixpoint recursion at the level of terms and recursion via lazy lists at the level of types. More specifically, we have shown that programs in such calculi which have undergone short cut fusion are contextually equivalent to their unfused counterparts. Our result formalizes the conventional wisdom concerning short cut fusion for these calculi.

The proof of the correctness of short cut fusion given here can be generalized to prove the correctness of generalizations of Gill's `foldr-augment` fusion (Gill, 1996) for versions of PolyPCF supporting algebraic data types other than lists. Specializing this result yields correctness proofs for short cut fusion for non-list algebraic data types, as well as `foldr-augment` fusion for lists, in these calculi. These results are detailed in Johann (2001).

Acknowledgements

I am grateful to Olaf Chitil, Graham Hutton and Andrew Pitts for helpful discussions on the topic of this paper. I also thank the anonymous referees and the editor for their comments and suggestions for improvement. This work was completed while visiting the Foundations of Programming group at the University of Nottingham. It was supported, in part, by the National Science Foundation under grant CCR-9900510.

References

Abadi, M. (2000) $\top\top$-closed relations and admissibility. *Mathematical Structures in Comput. Sci.*, **10**: 313–320.

Bainbridge, E. S., Freyd, P. J., Scedrov, A. and Scott, P. J. (1990) Functorial polymorphism. *Theor. Comput. Sci.*, **70**(1): 35–64. (Corrigendium in **71**(3): 431, 1990.)

Chitil, O. (1999) Type inference builds a short cut to deforestation. *Proceedings International Conference on Functional Programming*, pp. 249–260.

Gill, A. (1996) *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University.

Gill, A., Launchbury, J. and Peyton Jones, S. L. (1993) A short cut to deforestation. *Proceedings Conference on Functional Languages and Computer Architecture*, pp. 223–232.

Johann, P. (1997) An implementation of warm fusion. (Available at `ftp://ftp.cse.ogi.edu/pub/pacsoft/wf/`.)

Johann, P. (2001) Short cut fusion: Proved and improved. *Proceedings Workshop on Semantics, Applications and Implementation of Program Generation: Lecture Notes in Computer Science 2196*, pp. 47–71. Springer-Verlag.

Johann, P. and Visser, E. (2000) Warm fusion in Stratego: a case study in generation of program transformation systems. *Annals of Mathematics & Artif. Intell.*, **29**(1–4): 1–34.

Németh, L. (2000) *Catamorphism Based Program Transformations for Non-strict Functional Languages*. PhD thesis, Glasgow University.

Pitts, A. (1998) Existential types: Logical relations and operational equivalence. *Proceedings International Colloquium on Automata, Languages and Programming: Lecture Notes in Computer Science 1443*, pp. 309–326.

Pitts, A. (1998) Parametric polymorphism, recursive types, and operational equivalence. Unpublished Manuscript.

Pitts, A. (2000) Parametric polymorphism and operational equivalence. *Mathematical Structures in Comput. Sci.*, **10**: 1–39.

Takano, A. and Meijer, E. (1995) Shortcut deforestation in calculational form. *Proceedings Conference on Functional Programming and Computer Architecture*, pp. 324–333.

Reynolds, J. C. (1983) Types, abstraction, and parametric polymorphism. *Infor. Process.*, **83**: 513–523.

Sheard, T. and Fegaras, L. (1993) A fold for all seasons. *Proceedings Conference on Functional Programming and Computer Architecture*, pp. 233–242.

Wadler, P. (1989) Theorems for free! *Proceedings Conference on Functional Programming and Computer Architecture*, pp. 347–359.