Short Cut Fusion: Proved and Improved

By: Patricia Johann

Abstract

Short cut fusion is a particular program transformation technique which uses a single, local transformation — called the foldrbuild rule — to remove certain intermediate lists from modularly constructed functional programs. Arguments that short cut fusion is correct typically appeal either to intuition or to "free theorems" — even though the latter have not been known to hold for the languages supporting higher-order polymorphic functions and fixed point recursion in which short cut fusion is usually applied. In this paper we use Pitts' recent demonstration that contextual equivalence in such languages is relationally parametric to prove that programs in them which have undergone short cut fusion are contextually equivalent to their unfused counterparts. The same techniques in fact yield a much more general result. For each algebraic data type we define a generalization augment of build which constructs substitution instances of its associated data structures. Together with the well-known generalization cata of folder to arbitrary algebraic data types, this allows us to formulate and prove correct for each a contextual equivalence-preserving cata-augment fusion rule. These rules optimize compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them.

Patricia Johann (2001) "Short Cut Fusion: Proved and Improved". *Proceedings, Workshop on the Semantics, Application, and Implementation of Program Generation* (SAIG'01), Springer-Verlag LNCS 2196 (2001), pp. 47-71. Version Of Record Available At www.springer.com

1 Introduction

Fusion [4,5,6,17,15] is the process of removing certain intermediate data structures from modularly constructed functional programs. *Short cut fusion* [4,5] is a particular fusion technique which uses a single, local transformation rule — called the `foldr-build` rule — to fuse compositions of list-processing functions. The `foldr-build` rule is so named because it requires list-consuming and -producing functions to be written in terms of the program constructs `foldr` and `build`, respectively.

Short cut fusion successfully fuses a wide variety of list-processing programs, but its applicability is limited because list-producing functions cannot always be usefully expressed in terms of `build`. This observation led Gill [4] to introduce another construct, called `augment`, which generalizes `build` to efficiently handle more general list production. Gill also formulated a `foldr-augment` rule, similar to the `foldr-build` rule, for lists.

Like other fusion techniques, short cut fusion was first investigated for lists. This quickly gave rise to generalizations of short cut fusion for non-list algebraic data types, as well as to their incorporation into a number of automatic fusion tools (*e.g.*, [3,4,7,8,9,10]. Generalizations of `augment` and the `foldr-augment` rule for lists to non-list algebraic data types, on the other hand, have remained virtually unstudied.

In this paper we generalize Gill's `augment` for lists to non-list algebraic data types. Together with the well-known generalization `cata` of `foldr` to arbitrary algebraic data types, this allows us to formulate and prove correct for each a corresponding `cata-augment` fusion rule which generalizes the `foldr-augment` rule for lists. We interpret `augment` as constructing substitution instances of algebraic data structures, and view generalized `cata-augment` fusion as optimizing compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them.

1.1 The Problem of Correctness

Short cut fusion and its generalizations have successfully been used to improve programs in modern functional languages. They have even been used to transform modular programs into monolithic counterparts exhibiting order-of-magnitude efficiency increases over those from which they are derived. Nevertheless, there remain difficulties associated with their use. One of the most substantial is that these fusion techniques have not been proved correct for the languages in which they are typically applied.

Short cut fusion and its generalizations have traditionally been treated purely syntactically, with little consideration given to the underlying semantics of the languages in which they are applied. In particular, the fact that these fusion techniques are valid only for languages admitting parametric models has been downplayed in the literature. Instead, their application to functional programs has been justified by appealing either to intuition about the operational behavior of `cata`, `build`, and `augment`, or else to Wadler's "free theorems" [18].[1] But intuition is unsuitable as a basis for proofs, and the correctness of the "free theorems" itself relies on the existence of relationally parametric models. Since no relationally parametric models for modern functional languages are known to exist, these justifications of short cut fusion and its generalizations are unsatisfactory.

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 uniformity conditions are satisfied by models supporting a relationally parametric structure. Relationally parametric models are known to exist for some higher-order polymorphic languages [2], but because these fail to model fixed point recursion they

[1] In fact, the only proof of correctness of short cut fusion for a modern functional language on record [4] appeals to Wadler's "free theorems". Correctness proofs for generalizations of short cut fusion do not appear in the literature at present.

do not adequately accommodate short cut fusion and its generalizations. While it may be possible to extend these models to encompass fixed point recursion, this has not been reported in the literature. In fact, until recently the existence of relationally parametric models for languages supporting both higher-order polymorphic functions and fixed point recursion had not been demonstrated. Like their counterparts for the modern functional languages which extend them, short cut fusion and its generalizations for even the most streamlined of higher-order polymorphic languages with fixed point recursion have therefore enjoyed no proof of correctness.

1.2 Proving Correctness

In Section 5 of this paper we prove the correctness of generalized `cata-augment` fusion for algebraic data types in calculi supporting both higher-order polymorphic functions and fixed point recursion. Correctness for these calculi of short cut fusion for lists, the `foldr-augment` rule for lists, and generalizations of short cut fusion for lists to `cata-build` fusion for arbitrary algebraic data types, all are immediate consequences of this result. But because functional languages typically support features that cannot be modeled in the calculi considered here, our results do not apply to them directly. Nevertheless, our results do make some progress 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 fixed point recursion at the level of terms and recursion via data types with non-strict constructors at the level of types [13,14]. Pitts uses logical relations to characterize contextual equivalence in these 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 1) employs Pitts' characterization of contextual equivalence to demonstrate that programs in these calculi which have undergone generalized `cata-augment` fusion are contextually equivalent to their unfused counterparts. The semantic correctness of `cata-augment` 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 — *e.g.*, fixed point induction — 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 in which types are quantified only at the outermost level — can be achieved by appropriately restricting the characterizations for the corresponding impredicative ones.

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

The remainder of this paper is organized as follows. Section 2 informally introduces short cut fusion and `foldr-augment` fusion for lists. In Section 3 the polymorphic lambda calculus PolyFix for which we will formulate and prove the correctness of generalized `cata-augment` fusion is introduced; the notion of PolyFix contextual equivalence on which this relies is also formulated in Section 3. In Section 4, `cata-augment` fusion for arbitrary algebraic data types is formalized, and its correctness is proved in Section 5. Section 6 concludes.

2 Fusion

In functional programming, large programs are often constructed 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` 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 the informal discussion in this section 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. The above implementation of `mappend`, for instance, unnecessarily constructs and then traverses the intermediate list resulting from appending `xs` and `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. But 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.

2.1 Short Cut Fusion

Automatic elimination of intermediate data structures combines the clarity and maintainability of the modular style of programming with the efficiency of the monolithic style. Use of short cut fusion to eliminate intermediate lists is based on the observation that many list-manipulating functions can be written in terms of the list-consuming function `foldr` and the list-producing function `build`, and then fused via the `foldr-build` rule. Since `foldr` is another name for the standard catamorphism for lists, we denote it by `cata-list` in the rest of this paper. And since the `build` function of Gill *et al.* is the instantiation to lists of a `build` function applying to more general algebraic data types, we denote it by `build-list` below.

Operationally, `cata-list` takes as input types `t` and `t'`, a replacement term `c :: t -> t' -> t'` for `Cons`, a replacement term `n::t'` for `Nil`, and a list `xs`

```
map :: forall a. forall b. (a -> b) -> List a -> List b
map = /\a b. \f l. build-list b
                   (/\t. \(c :: b -> t -> t) (n::t).
                      cata-list a t
                        (\(y::a) (l'::t). c (f y) l') n l)
append :: forall a. List a -> List a -> List a
append = /\a. \xs ys. build-list a
                      (/\t. \(c::a -> t -> t) (n::t).
                         cata-list a t c
                           (cata-list a t c n ys) xs)
```

Fig. 2. Functions in build-cata form

of type `List t`. It replaces all (fully-applied) occurrences of `Cons` in `xs` by `c`, and the single occurrence of `Nil` in `xs` by `n`. The result is a value of type `t'`. The definition of `cata-list` — *i.e.*, of `foldr` — appears in Figure 1.

The function `build-list`, 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 `Cons` and `Nil`. The result is a list of elements of type `t`. That is, if `t` is a type and `M` is any term with type `forall a. (t -> a -> a) -> a -> a`, then

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

Compositions of list-consuming and -producing functions defined in terms of `cata-list` and `build-list` can be fused via *short cut fusion* for lists:

> Let `M` be a term of type `forall a. (t -> a -> a) -> a -> a`. Then any occurrence of `cata-list t t' c n (build-list t M)` in a program can be replaced by `M t' c n`.

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`, `Cons`, and `Nil` and then these are replaced in the resulting list by `c` and `n`, respectively, or the abstract constructors in (an appropriate instance of) `M` are replaced by `c` and `n`, respectively, directly.

Figure 2 shows the `build-cata` forms of the functions in 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 simplifications.

2.2 The Cata-Augment Rule for Lists

Although short cut fusion successfully fuses many compositions of list-processing functions, some compositions involving common functions remain problematic. This is because the argument `M` to `build-list` must abstract *all* of the `Cons` and

`Nil` cells which appear in the list it produces — not just the "top-level" ones contributed by `M` itself.

To see why, suppose that we want to express the function `append` for lists of elements of an arbitrary type `t` in terms of `build-list` and `cata-list`. This would make it possible to fuse `append` with list-producers on the right and list-consumers on the left. It is tempting to write

```
append = /\a. \xs ys. build-list a
                        (/\t. \c n. cata-list a t c ys xs)
```

but the expression on the right hand side is ill-typed: `ys` is of type `List a`, but `cata-list`'s replacement for `Nil` needs to be of the more general type `t`. The problem here is that, although the constructors in `ys` are part of the result of `append`, they are not properly abstracted by `build-list`.

One solution to this problem is to use `cata-list` to prepare the constructors in `ys` for abstraction via `build-list`. This entails replacing the occurrence of `ys` in the body of the definition of `append` by `cata a t c ys`. The result is the `build-cata` form

```
append = /\a. \xs ys. build-list a (/\t. \c n.
                        cata-list a t c (cata-list a t c ys) xs)
```

for `append`. Although this solution does indeed provide a replacement of type `t` for `Nil`, it does so by introducing an extra list consumption into the computation. Unfortunately, subsequent removal of this consumption via fusion cannot be guaranteed.

An alternative solution is to generalize `build-list` to abstract the "base list" `ys` of `append`. Gill *et al.* [5] adopt this approach, defining a new construct `augment-list` to perform this task. Its definition is

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

Appending one list onto another is now easily expressed by passing `ys` as the second argument to `augment-list`:

```
append = /\a. \xs ys. augment-list t
                         (/\t. \c n. cata-list a t c n xs) ys
```

This definition of `augment-list` also gives

```
build-list t M = augment-list t M Nil
```

and is the basis for *the* `cata-augment` *rule* for lists from Gill [4] for fusing compositions of functions written in terms of `cata-list` and `augment-list`:

> Let `t` be a type, let `M :: forall a. (t -> a -> a) -> a -> a` be a closed term, and let `ys :: List t`. Then any occurrence of
>
> ```
> cata-list t t' c n (augment-list t M ys)
> ```
>
> in a program can be replaced by

```
M t' c (cata-list t t' c n ys).
```

The short cut for lists is just the special case of the `cata-augment` rule for lists in which `ys` has been specialized to `Nil`, `augment-list` has been replaced by `build-list`, and `cata-list t t' c n` has been applied to `Nil` to yield `n`. Although the `cata-augment` rule for lists does not eliminate the entire intermediate list produced by the left-hand side of the equation, it does avoid production and subsequent consumption of the base list `ys`. Passing `ys` to `augment-list` has the effect of specifying a particular list to be substituted for the occurrence of `Nil` in the list produced by `M`. This interpretation of `augment` as constructing substitution instances of data structures will lead to generalizations of `augment-list` and the `cata-augment` rule for lists to algebraic data types in Section 4.

3 PolyFix and Contextual Equivalence

Extrapolating from the situation for lists, we formulate in Section 4 a suitable generalization of the `cata-augment` rule for lists to one for non-list algebraic data types. In Section 5 we demonstrate that these generalizations describe optimizations for the uniform consumption of uniformly produced substitution instances of non-list algebraic data structures. In doing so, we work in the same setting as Pitts [13], and our presentation is heavily influenced by that paper. In this section we introduce Pitts' PolyFix, the polymorphic lambda calculus for which we state, and prove the correctness of, the generalized `cata-augment` rule. We also outline the elements of Pitts' development of contextual equivalence for terms of PolyFix which are needed in this endeavor.

3.1 PolyFix: The Fixed Point Calculus

The *Polymorphic Fixed Point Calculus* PolyFix combines the Girard-Reynolds polymorphic lambda calculus with fixed point recursion at the level of expressions and (positive) recursion via non-strict constructors at the level of 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 algebraic data types, for notational convenience we assume that PolyFix supports only the latter.

The syntax of PolyFix types and terms is given in Figure 3, in which the Haskell-like syntax

$$\mathtt{data}(\alpha \;=\; \mathsf{c}_1^\delta \tau_{11} ... \tau_{1k_1} \mid ... \mid \mathsf{c}_m^\delta \tau_{m1} ... \tau_{mk_m}) \tag{1}$$

is used for recursive data types. The syntax in (1) provides an anonymous notation for a data type δ satisfying the fixed point equation

$$\delta = (\tau_{11}[\delta/\alpha] \times ... \times \tau_{1k_1}[\delta/\alpha]) + ... + (\tau_{m1}[\delta/\alpha] \times ... \times \tau_{mk_m}[\delta/\alpha])$$

The injections into the m-fold sum are named explicitly by δ's constructors c_1^δ,...,c_m^δ. Terms of type δ are introduced using these constructors and eliminated

Types	τ	$:= \alpha$	type variable
		$\mid \tau \to \tau$	function type
		$\mid \forall\alpha.\tau$	$\forall$-type
		$\mid \delta$	algebraic data type
Data types	δ	$:= \texttt{data}(\alpha \texttt{ = } \mathtt{c}_1^\delta \overline{\tau_{k_1}} \mid ... \mid \mathtt{c}_m^\delta \overline{\tau_{k_m}})$	
Terms	M	$:= x$	variable
		$\mid \lambda x : \tau.\, M$	function abstraction
		$\mid MM$	function application
		$\mid \Lambda\alpha.\, M$	type abstraction
		$\mid M\tau$	type application
		$\mid \texttt{fix}(M)$	fixpoint recursion
		$\mid \mathtt{c}_i^\delta \overline{M_{k_i}}$	data value
		$\mid \texttt{case } M \texttt{ of}$	
		$\quad \{\mathtt{c}_1^\delta \overline{x_{k_1}} \Rightarrow M \mid$	
		$\quad\quad ...$	
		$\quad\quad \mid \mathtt{c}_m^\delta \overline{x_{k_m}} \Rightarrow M\}$	case expression

Fig. 3. Syntax of PolyFix

using case expressions. The types τ_{ij}, for $i = 1, ..., m$ and $j = 1, ..., k_i$, appearing in (1) can be built up from type variables using function types, $\forall$-types, and data types, provided the defined type α occurs only positively in the τ_{ij} (see Definition 1 below).

Example 1. The following are PolyFix data types:

$$\begin{aligned} \mathit{Bool} &= \texttt{data}(\alpha = \texttt{True} \mid \texttt{False}) \\ \mathit{Nat} &= \texttt{data}(\alpha = \texttt{Succ } \alpha \mid \texttt{Zero}) \\ \mathit{List}\,\tau &= \texttt{data}(\alpha = \texttt{Cons } \tau\ \alpha \mid \texttt{Nil}) \end{aligned}$$

A number of remarks concerning the definitions of Figure 3 are in order. Type variables, variables, and constructors range over disjoint countably infinite sets. If s ranges over a set S, then for each n, $\overline{s_n}$ ranges over n-element sequences of elements of S. If M is a term and $\overline{s_n}$ is a sequence of n types or terms, we write $M\overline{s_n}$ to indicate the n-fold application $Ms_1...s_n$. Similarly, we write $\lambda\overline{x_n} : \overline{\tau_n}.\, M$ to indicate the n-fold abstraction $\lambda x_1 : \tau_1.\ ...\lambda x_n : \tau_n.\, M$. Finally, to be well-formed, we require a data type as in (1) to have distinct data constructors $\mathtt{c}_i^\delta$, $i = 1, ..., m$, and to be algebraic in the sense of Definition 1.

Definition 1. *The sets $ftv^+(\tau)$ and $ftv^-(\tau)$ of free type variables* occurring positively *and* occurring negatively *in the type τ partition $ftv(\tau)$ into two disjoint subsets. These are defined by*

$$
\begin{array}{ll}
ftv^{+}(\alpha) & = \{\alpha\} \\
ftv^{-}(\alpha) & = \emptyset \\
ftv^{\pm}(\tau \to \tau') & = ftv^{\mp}(\tau) \cup ftv^{\pm}(\tau') \\
ftv^{\pm}(\forall\alpha.\,\tau) & = ftv^{\pm}(\tau) \setminus \{\alpha\} \\
ftv^{\pm}(\delta) & = \bigcup_{i=1}^{m} \bigcup_{j=1}^{k_m} ftv^{\pm}(\tau_{ij}) \setminus \{\alpha\} \text{ if } \delta \text{ is as in (1).}
\end{array}
$$

A data type (1) is algebraic *if there are only positive free occurrences of its bound variable α in the types τ_{ij},* i.e.*, if $\alpha \notin ftv^{-}(\tau_{ij})$ for all $i = 1, ..., m$ and $j = 1, ..., k_i$.*

The constructions $\forall\alpha(-)$, $\mathtt{data}(\alpha = -)$, $\mathtt{case}\ M\ \mathtt{of}\ \{... \mid \mathtt{c}_i^{\delta}\overline{x_{k_i}} \Rightarrow M_i \mid ...\}$, $\lambda x : \tau.-$, and $\Lambda\alpha.-$ are binders, and free occurrences of the variables $x_1, ..., x_{k_i}$ become bound in the case expression $\mathtt{case}\ D\ \mathtt{of}\ \{... \mid \mathtt{c}_i^{\delta}\overline{x_{k_i}} \Rightarrow M_i \mid ...\}$. 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 $M[M'/x]$.

We will be concerned only with PolyFix terms which are typeable. The type assignment relation for PolyFix is completely standard; it is given in Figure 4. In the last two clauses of Figure 4, δ is assumed to be $\mathtt{data}(\alpha = \mathtt{c}_1\overline{\tau_{1k_1}} \mid ... \mid \mathtt{c}_m\overline{\tau_{mk_m}})$. A *typing environment* Γ is a pair A, Δ with A a finite set of type variables and Δ 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 extension of A with a type variable $\alpha \notin A$.

The explicit type annotations on lambda-bound term variables and constructors $\mathtt{c}_i^{\delta}$ in data values $\mathtt{c}_i^{\delta}\overline{M_{k_i}}$ ensure that well-formed PolyFix terms have unique types. More specifically, given Γ and M, there is at most one type τ for which $\Gamma \vdash M : \tau$ holds. For convenience we may 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 PolyFix types is denoted *Typ*. For $\tau \in Typ$ the set of closed PolyFix terms M for which $\emptyset, \emptyset \vdash M : \tau$ is denoted $Term(\tau)$.

Given δ as in (1), let Rec_{δ} comprise the elements i of $\{1, ..., m\}$ for which $\alpha_{ij} \in fv(\tau_{ij})$ for some $j \in \{1, ..., k_i\}$, and let $NonRec_{\delta}$ be the set $\{1, ..., m\} - Rec_{\delta}$. We say that the data constructors $\mathtt{c}_i$, $i \in Rec_{\delta}$, are *recursive constructors* of δ and that $\mathtt{c}_i$, $i \in NonRec_{\delta}$, are *nonrecursive constructors* of δ. In addition, given a constructor $\mathtt{c}_i$, let $RecPos_{\mathtt{c}_i}$ comprise those elements $j \in \{1, ..., k_i\}$ for which $\alpha \in fv(\tau_{ij})$, and let $NonRecPos_{\mathtt{c}_i}$ be the set $\{1, ..., k_i\} - RecPos_{\mathtt{c}_i}$. We say that the indices in $RecPos_{\mathtt{c}_i}$ indicate the *recursive positions* of $\mathtt{c}_i$ and

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

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

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

$$\frac{\Gamma \vdash \mathtt{M}_j:\tau_j[\delta/\alpha] \qquad j=1,..,k_i}{\Gamma \vdash \mathtt{c}_i M_1 ... M_{k_i}:\delta}$$

$$\frac{\Gamma \vdash D:\delta \qquad \Gamma,\, \overline{x_{k_i}}:\overline{\tau_{k_i}[\delta/\alpha]} \vdash M_i:\tau \qquad i=1,..,m}{\Gamma \vdash \mathtt{case}\ D\ \mathtt{of}\ \{\mathtt{c}_1\overline{x_{k_1}} \Rightarrow M_1 \mid ... \mid \mathtt{c}_m\overline{x_{k_m}} \Rightarrow M_m\}:\tau}$$

Fig. 4. PolyFix type assignment

that the indices in $NonRecPos_{\mathtt{C}_i}$ indicate the *nonrecursive positions* of $\mathtt{c}_i$. The distinction between recursive and nonrecursive constructors and positions will be useful to us in stating our main result in Section 4.

The notation of the next definition allows us to order, and to project onto the resulting sequence of, arguments to function abstractions. We will use it to express `cata`, `build`, and `augment` in PolyFix.

Definition 2. *Let δ be as in (1), τ be a closed type, $Rec_\delta = \{u_1, ..., u_p\}$, and $NonRec_\delta = \{v_1, ..., v_q\}$. For all $\rho_i : \tau_{i1}[\tau/\alpha] \to ... \to \tau_{ik_i}[\tau/\alpha] \to \tau$, for all $i = 1, ..., m$, define*

$$\phi_i(\rho_{u_1}, ..., \rho_{u_p}, \rho_{v_1}, ..., \rho_{v_q}) = \rho_i.$$

Further, if $RecPos_{\mathtt{C}_i} = \{z_1, ..., z_p\}$ and $NonRecPos_{\mathtt{C}_i} = \{y_1, ..., y_q\}$ for all $i = 1, ..., m$, define

$$\phi_{ij}(\rho_{z_1}, ..., \rho_{z_p}, \rho_{y_1}, ..., \rho_{y_q}) \;=\; \rho_j.$$

For each data type δ as in (1) we can also define a corresponding pure polymorphic type τ_δ by

$$\tau_\delta = \forall\alpha.(\tau_{11} \to ... \to \tau_{1k_1} \to \alpha) \to ... \to (\tau_{m1} \to ... \to \tau_{mk_m} \to \alpha) \to \alpha.$$

Using these types, we have

$$V \Downarrow V \ (V \text{ is a value}) \qquad \frac{F \Downarrow \lambda x : \tau.\, M \qquad M[A/x] \Downarrow V}{F\, A \Downarrow V}$$

$$\frac{G \Downarrow \Lambda\alpha.\, M \qquad M[\tau/\alpha] \Downarrow V}{G\, \tau \Downarrow V} \qquad \frac{F\ (\mathtt{fix}\, F) \Downarrow V}{\mathtt{fix}\, F \Downarrow V}$$

$$\frac{D \Downarrow \mathtt{c}_i \overline{M_{k_i}} \qquad M[\overline{M_{k_i}}/\overline{x_{k_i}}] \Downarrow V}{\mathtt{case}\ D\ \mathtt{of}\ \{... \mid \mathtt{c}_i \overline{x_{k_i}} \Rightarrow M \mid ...\} \Downarrow V}$$

Fig. 5. PolyFix evaluation relation

Definition 3. *For each data type δ define*

$$\begin{aligned}
\mathtt{build}^{\delta} &= \lambda M : \tau_{\delta}.\, M\ \delta\ \overline{c_m} \\
&\quad \textit{where for each } \mathtt{c}_i, \textit{ we define } c_i = \lambda \overline{p_{k_i}} : \overline{\tau_{k_i}[\delta/\alpha]}.\, \mathtt{c}_i \overline{p_{k_i}} \\
\mathtt{unbuild}^{\delta} &= \mathtt{fix}(\lambda h : \delta \to \tau_{\delta}.\, \lambda d : \delta.\, \Lambda\alpha.\, \lambda \overline{f_m} : \overline{\tau_{m1} \to ... \to \tau_{mk_m} \to \alpha}. \\
&\quad\ \mathtt{case}\ d\ \mathtt{of} \\
&\qquad \{... \mid \mathtt{c}_i \overline{x_{k_i}} \Rightarrow f_i \overline{\phi_{ik_i}(h x_{z_1} \alpha \overline{f_m}, ... h x_{z_p} \alpha \overline{f_m}, x_{y_1}, ..., x_{y_q})} \mid ...\}) \\
&\quad \textit{where } RecPos_{\mathtt{c}_i} = \{z_1, ..., z_p\} \textit{ and } NonRecPos_{\mathtt{c}_i} = \{y_1, ..., y_q\} \\
\mathtt{cata}^{\delta} &= \Lambda\alpha.\ \lambda \overline{f_m}.\ \lambda d.\, \mathtt{unbuild}^{\delta}\ d\ \alpha\ \overline{f_m}
\end{aligned}$$

If δ is closed then each of $\mathtt{build}^{\delta}$, $\mathtt{unbuild}^{\delta}$, and $\mathtt{cata}^{\delta}$ is a closed PolyFix term. In the notation of Definition 3, we have that $\mathtt{cata\text{-}list}\ \tau = \mathtt{cata}^{List\,\tau}$ and $\mathtt{build\text{-}list}\ \tau = \mathtt{build}^{List\,\tau}$. Note that data type constructors must be fully applied in well-formed PolyFix terms.

3.2 Operational Semantics

The operational semantics of PolyFix is given by the *evaluation relation* in Figure 5. There, δ is assumed to be $\mathtt{data}(\alpha = ... \mid \mathtt{c}_i \overline{\tau_{ik_i}} \mid ...))$ It relates a closed term M to a value V of the same closed type; this is denoted $M \Downarrow V$. The set of PolyFix *values* is given by

$$V ::= \lambda x : \tau.M \mid \Lambda\alpha.M \mid \mathtt{c}_i \overline{M_{k_i}}$$

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, PolyFix evaluation is deterministic, although the rule for `fix` entails the existence of terms whose evaluation does not terminate.

3.3 Contextual Equivalence

With the operational semantics of PolyFix 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 change in observable behavior when the resulting programs are executed. In order to formalize this notion for PolyFix we must specify what a PolyFix program is, as well as the PolyFix program behavior we are interested in observing.

We define a PolyFix *program* to be a closed term of some data type, and the *observable behavior* of a PolyFix program to be the outermost constructor in the data value, if any, to which the program evaluates. (Recall that ground types have been replaced by algebraic data types in PolyFix.) Since merely observing termination of PolyFix evaluation (or lack thereof) at data types gives rise to the same notion of contextual equivalence, we define two PolyFix 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 $\mathcal{M}[-]$ for which $\mathcal{M}[M_1], \mathcal{M}[M_2] \in \mathit{Term}(\delta)$ for some closed data type δ, we have

$$\mathcal{M}[M_1] \Downarrow \Leftrightarrow \mathcal{M}[M_2] \Downarrow .$$

As usual, a *context* $\mathcal{M}[-]$ is a PolyFix term with a subterm replaced by the placeholder '$-$', and $\mathcal{M}[M]$ denotes the term which results from replacing the placeholder by the term M. Note that replacement may involve variable capture. We write $\Gamma \vdash M_1 =_{ctx} M_2 : \tau$ to indicate that M_1 and M_2 are contextually equivalent with respect to Γ. If M_1 and M_2 are closed terms of closed type, we write $M_1 =_{ctx} M_2 : \tau$ instead of $\emptyset, \emptyset \vdash M_1 =_{ctx} M_2 : \tau$, and we say simply that M_1 and M_2 are *contextually equivalent*.

For all terms M and M' of type τ_1, A of type τ_2, and F of type τ, the following contextual equivalences are shown to hold in Pitts [13]:

$$(\lambda x : \tau_2.\, M)A =_{ctx} M[A/x] : \tau_1 \tag{2}$$

$$(\Lambda\alpha.\, M)\tau_2 =_{ctx} M[\tau_2/\alpha] : \tau_1[\tau_2/\alpha] \tag{3}$$

$$\texttt{case}\ \mathtt{c}_i^{\delta}\overline{M_{k_i}}\ \texttt{of}\ \{... \mid \mathtt{c}_i^{\delta}\overline{x_{k_i}} \Rightarrow M' \mid ...\} =_{ctx} M'[\overline{M_{k_i}}/\overline{x_{k_i}}] : \tau_1 \tag{4}$$

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

4 A Generalized Cata-Augment Rule

In this section we state our main result, the Substitution Theorem. This theorem allows us to generalize Gill's `cata-augment` rule for lists to arbitrary algebraic data types. It also allows us to make precise the sense in which the generalized `cata-augment` rule and its specializations preserve the meanings of fused PolyFix programs. Proof of the Substitution Theorem appears in Section 5.3.

We will consider only closed types and terms in the remainder of this paper. This restriction is reasonable because contextual equivalence for open terms is reducible to contextual equivalence for closed terms, as shown in Pitts [13].

Theorem 1. (Substitution Theorem) *Let δ be a closed data type as in (1), and let $u_1, ..., u_p, v_1, ..., v_q$, and $\phi_1, ..., \phi_m$ be as in Definition 2. In addition, let*

$$M : \forall\alpha.(\tau_{11} \to ... \to \tau_{1k_1} \to \alpha) \to ... \to (\tau_{m1} \to ... \to \tau_{mk_m} \to \alpha) \to \alpha$$

be a closed term, let τ, $\tau'_{ij} = \tau_{ij}[\tau/\alpha]$, and $\tau''_{ij} = \tau_{ij}[\delta/\alpha]$ for $i = 1, ..., m$ and $j = 1, ..., k_i$ be closed types, and, for $i = 1, ..., m$ and $v \in NonRec_\delta$, let

$$c_i \;=\; \lambda\overline{p_{k_i}} : \overline{\tau''_{k_i}}.\, \mathtt{c}_i\ \overline{p_{k_i}},$$

$$n_i : \tau'_{i1} \to ... \to \tau'_{ik_i} \to \tau,$$

and

$$\mu_v : \tau''_{v1} \to ... \to \tau''_{vk_v} \to \delta$$

be closed terms. Then

$$\begin{aligned} &\mathtt{cata}^\delta\ \tau\ \overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\ \ (M\ \delta\ \overline{\phi_m(c_{u_1}, ...c_{u_q}, \mu_{v_1}, ..., \mu_{v_q})}) \\ =_{ctx}\ &M\ \tau\ \overline{\phi_m(n_{u_1}, ..., n_{u_p}, \mu'_{v_1}, ..., \mu'_{v_q})}\ :\ \tau \end{aligned}$$

where, for each $v \in NonRec_\delta$, the closed term $\mu'_v : \tau'_{v1} \to ... \to \tau'_{vk_v} \to \tau$ is given by

$$\mu'_v x_1 ... x_{k_v} = \mathtt{cata}^\delta\ \tau\ \overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\ (\mu_v x_1 ... x_{k_v}).$$

The functions μ_v for $v \in NonRec_\delta$ can be thought of as substitutions mapping appropriate combinations of arguments of types τ''_{vj}, $j = 1, ..., k_v$, to terms of type δ; they determine the portion of the intermediate data structure not produced by M itself, *i.e.*, the non-initial segment of the intermediate data structure. The Substitution Theorem describes one way to optimize uniform consumption of substitution instances of algebraic data structures: it says that the result of using μ_v to substitute terms of data type δ for applications of the nonrecursive data constructors in a uniformly produced element of type δ, and then consuming the data structure resulting from that substitution with a catamorphism, is the same as simply producing the "abstract" data structure in which applications of recursive data constructors are replaced by their corresponding arguments to the catamorphism, and nonrecursive data constructors are replaced by the results of applying the catamorphism to their substitution values.

Just as the `cata-augment` rule for lists avoids production and then consumption of the portion of the intermediate list constructed by `augment`'s polymorphic function argument, so the Substitution Theorem indicates how to avoid production and subsequent consumption of the initial segments of more general algebraic data structures. Additional efficiency gains may be achieved in situations in which the representations of the substitutions μ_v allow us to carry out each application $\mathtt{cata}^\delta\ \overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\ (\mu_v x_1 ... x_{k_v})$ exactly once.

If we generalize the definition of `augment-list` to a non-list data type δ by

$$\mathtt{augment}^\delta = \lambda M.\, \lambda\overline{\mu_{v_q}}.\, M\ \delta\ \overline{\phi_m(c_{u_1}, ..., c_{u_p}, \mu_{v_1}, ..., \mu_{v_q})}$$

then we can use this notation to rephrase the conclusion of Theorem 1 in a manner reminiscent of the `cata-augment` rule for lists:

Definition 4. *Suppose the conditions of Theorem 1 hold. The* generalized cata-augment rule *is given by*

$$\begin{aligned}&\texttt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\ \ (\texttt{augment}^{\delta}\ M\ \overline{\mu_{v_q}})\\ =_{ctx}\ &M\,\tau\,\overline{\phi_m(n_{u_1},...,n_{u_p},\mu'_{v_1},...,\mu'_{v_q})}\ :\ \tau\end{aligned}$$

Note that the generalized `cata-augment` rule allows the replacement terms for the nonrecursive data constructors to be specified by any appropriately typed substitutions $\mu_{v_1},...,\mu_{v_q}$. In this notation, the `cata-augment` rule for lists requires exactly one such term, corresponding to the nonrecursive constructor `Nil`. Since `Nil` takes no term arguments we see that, for each type τ, each term $M : \tau_\delta$, and each $\mu : List\,\tau$, `augment-list` $\tau\,M\,\mu$ from Section 2.2 is precisely $\texttt{augment}^{List\,\tau}\,M\,\mu$.

For any data type δ, specializing μ_v to c_v for $v \in NonRec_\delta$ in $\texttt{augment}^{\delta}\,M\,\overline{\mu_{v_q}}$ gives $\texttt{build}^{\delta}\,M$, just as for lists. With this specialization, the Substitution Theorem yields the usual `cata-build` rule for algebraic data types. The Substitution Theorem thus makes precise the sense in which short cut fusion for these data types preserves the meanings of programs.

Note that the term arguments to `build` and `augment` need not be closed in function definitions; in fact, none of the term arguments to `build-list` in the definitions of Figure 2 are closed terms. While this observation may at first glance suggest that the generalized `cata-augment` rule and its specializations cannot be applied to them, in all situations in which these rules are used to fuse programs the free variables in the term arguments to `build` and `augment` will already have been instantiated with closed terms.

The following example illustrates the use of the generalized `cata-augment` rule to remove a non-list intermediate data structure from a program.

Example 2. Let $Expr\,\tau$, a data type of expressions, be given by

$$Expr\,\tau\ =\ \texttt{data}(\alpha = \texttt{Var}\,\tau \mid \texttt{Lit}\,Nat \mid \texttt{Op}\,Ops\,\alpha\,\alpha),$$

where

$$Ops\ =\ \texttt{data}(\alpha = \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mul} \mid \texttt{Div}).$$

Also let $env : \tau \to Expr\,\tau$ be an expression environment, let $\texttt{mu}_{\texttt{Var}} = env$, $\mu_{\texttt{Lit}} = \lambda i : Nat.\,\texttt{Lit}\,i$, and $c_{\texttt{Op}} = \lambda o : Ops.\,\lambda v_1 : Expr\,\tau.\lambda v_2 : Expr\,\tau.\,\texttt{Op}\,o\,v_1\,v_2$. Finally, for each $e : Expr\,\tau$, let

$$M_e : \forall\alpha.(\tau \to \alpha) \to (Nat \to \alpha) \to (\texttt{Ops} \to \alpha \to \alpha \to \alpha) \to \alpha$$

be the polymorphic function associated with e; for example,

$$\begin{aligned}M_{\texttt{Var}\,n} &= \forall\alpha.\,\lambda v.\,\lambda l.\,\lambda o.\,v\,n,\\ M_{\texttt{Lit}\,9} &= \forall\alpha.\,\lambda v.\,\lambda l.\,\lambda o.\,l\,9,\end{aligned}$$

and

$$\begin{aligned}&M_{\texttt{Op Add (Lit }7\texttt{)(Op Sub (Lit }4\texttt{) (Var }m\texttt{))}}\\ =\ &\forall\alpha.\,\lambda v.\,\lambda l.\,\lambda o.\,o(\texttt{Add}, l\,7, o(\texttt{Sub}, l\,4, v\,m)).\end{aligned}$$

Then

$$subst\ env\ e = \mathtt{augment}^{Expr\,\tau}\ M_e\ \mu_{\mathtt{Var}}\ \mu_{\mathtt{Lit}}$$

defines a substitution function over $Expr\,\tau$, and

$$eval = \mathtt{cata}^{Expr\,\tau}\ Nat\ error\ (\lambda i : Nat.\, i)\ (\lambda\, o.\lambda v_1.\, \lambda v_2.\, o\, v_1\, v_2)$$

defines an evaluation function $eval : Expr\,\tau \to Nat$. Here, *error* indicates a failed computation.

We can use the generalized `cata-augment` rule to optimize the evaluation of a substitution instance of an expression:

$$eval\ (subst\ env\ e) = M_e\ Nat\ \mu'_{\mathtt{Var}}\ \mu'_{\mathtt{Lit}}\ (\lambda\, o.\lambda v_1.\, \lambda v_2.\, o\, v_1\, v_2)$$

Here, $\mu'_{\mathtt{Var}} = \lambda n.\, eval\ (env\ n)$ and $\mu_{\mathtt{Lit}} = \lambda i : Nat.\, eval\,(\mathtt{Lit}\, i)$. The fusion performed gives a more efficient, yet functionally equivalent, implementation of evaluation of substitution instances of expressions in which substitution and evaluation are interleaved.

5 Correctness of Cata-Augment Fusion

To prove the Substitution Theorem we would like to define a logical relation which coincides with PolyFix contextual equivalence, while at the same time incorporating a notion of relational parametricity analogous to that introduced by Reynolds for the pure polymorphic lambda calculus [16]. Unfortunately, a naive approach to defining such a logical relation — *i.e.*, one which quantifies over *all* appropriately typed relations in the defining clause for $\forall$-types — is not sufficiently restrictive to give good parametricity behavior. What is needed is some criterion for identifying precisely those relations which are "admissible for fixpoint induction," in the sense that they syntactically capture to 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 [13], provides a criterion sufficient to guarantee this kind of admissibility [1].

The notion of $\top\top$-closure is induced by a Galois connection between term relations and evaluation contexts, *i.e.*, contexts $\mathcal{M}[-]$ which have a single occurrence of the placeholder '$-$' in the position at which the next subexpression will be evaluated. In Pitts [13], analysis of evaluation contexts is aided by recasting them in terms of the notion of frame stack given in Definition 5 below; indeed, this frame stack realization of evaluation contexts gives rise to Pitts' syntactic characterization of the PolyFix termination properties entailed by contextual equivalence. The resulting PolyFix *structural termination relation* provides the key to appropriately specifying 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 4.1 and 4.2, we use it in Section 5 to prove the Substitution Theorem.

$$\Gamma \vdash Id : \tau \hookrightarrow \tau$$

$$\frac{\Gamma \vdash S : \tau' \hookrightarrow \tau'' \qquad \Gamma \vdash A : \tau}{\Gamma \vdash S \circ (-M) : (\tau \hookrightarrow \tau') \hookrightarrow \tau''}$$

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

$$\frac{\Gamma \vdash S : \tau \hookrightarrow \tau' \qquad \Gamma, \overline{x_{k_i}} : \overline{\tau_{ik_i}} \vdash M_i : \tau \qquad i = 1, .., m}{\Gamma \vdash S \circ (\texttt{case} \ - \ \texttt{of} \ \{\texttt{c}_1 \overline{x_{1k_1}} \Rightarrow M_1 \mid ... \mid ...\texttt{c}_m \overline{x_{mk_m}} \Rightarrow M_m\}) : \delta \hookrightarrow \tau'}$$

Fig. 6. Frame stack type judgements

5.1 $\top\top$-closed Relations

Definition 5. *The grammar for PolyFix* frame stacks *is*

$$S ::= Id \mid S \circ F$$

where F ranges over frames*:*

$$F ::= (-M) \mid (-\tau) \mid \texttt{case} \ - \ \texttt{of} \ \{...\}.$$

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 occurring in the judgement. The axioms and rules inductively defining this judgement 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 use.

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

$$Stack(\tau) = \bigcup \{Stack(\tau, \delta) \mid \delta \text{ is a data type}\}$$

The operation $S, M \mapsto SM$ 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:

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

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

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

$$\frac{S \circ (-\mathtt{fix}\, F) \top F}{S' \top \mathtt{fix}\, F} \qquad \frac{S = Id}{S \top \mathtt{c}_i\overline{M_{k_i}}}$$

$$\frac{S = S' \circ \mathtt{case}\ -\ \mathtt{of}\ \{... \mid \mathtt{c}_i\overline{M_{k_i}} \Rightarrow M' \mid ...\} \qquad S' \top M'[\overline{M_{k_i}}/\overline{x_{k_i}}]}{S \top \mathtt{c}_i\overline{M_{k_i}}}$$

$$\frac{S \circ \mathtt{case}\ -\ \mathtt{of}\ \{...\} \top M}{S \top \mathtt{case}\ M\ \mathtt{of}\ \{...\}}$$

Fig. 7. PolyFix structural termination relation

Here, $F[M]$ is the term that results from replacing '$-$' by M in the frame F. If $S \in \mathit{Stack}(\tau, \tau')$ and $M \in \mathit{Term}(\tau)$, then $SM \in \mathit{Term}(\tau')$. Unlike PolyFix 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 PolyFix evaluation relation in Figure 5.

PolyFix termination is captured by the termination relation $(-)\top(-)$ defined in Figure 7. More precisely, for all closed types τ, all closed data types δ, all frame stacks $S \in \mathit{Stack}(\tau, \delta)$, and all $M \in \mathit{Term}(\tau)$,

$$S\, M \Downarrow \text{ iff } S \top M.$$

Pitts uses this characterization of PolyFix 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 6. *A PolyFix* 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 PolyFix* stack relation *is a binary relation between (typeable) frame stacks whose result types are data 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 7. *Given any closed types τ and τ', and any $r \in Rel(\tau, \tau')$, define $r^{\top} \in Rel^{\top}(\tau, \tau')$ by*

$$(S, S') \in r^{\top} \Leftrightarrow \forall (M, M') \in r.\, S \top M \Leftrightarrow S' \top M'$$

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

$$(M, M') \in s^{\top} \Leftrightarrow \forall (S, S') \in s.\, S \top M \Leftrightarrow S' \top M'$$

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 8. *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

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

This characterization of $\top\top$-closedness will be used in Section 5.3.

5.2 Characterizing Contextual Equivalence

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

Definition 9. Action of $\rightarrow$ on term relations: *Given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, define $r_1 \rightarrow r_2 \in Rel(\tau_1 \rightarrow \tau_2, \tau_1' \rightarrow \tau_2')$ by*

$$(F, F') \in r_1 \rightarrow r_2 \;\Leftrightarrow\; \forall (A, A') \in r_1.\, (FA, F'A') \in r_2$$

Action of $\forall$ on term relations: *Let τ_1 and τ_1' be types with at most one free type variable 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

$$(G, G') \in \forall r.\, R(r) \;\Leftrightarrow\; \forall \tau_2, \tau_2' \in Typ.\, \forall r \in Rel(\tau_2, \tau_2').\, (G\tau_2, G'\tau_2') \in R(r)$$

Action of data constructors on term relations: *Let δ and δ' be the closed data types*

$$\delta = \mathtt{data}(\alpha \;=\; \mathtt{c}_1\tau_{11}...\tau_{1k_1} \mid ... \mid \mathtt{c}_m\tau_{m1}...\tau_{mk_m})$$

and

$$\delta' = \mathtt{data}(\alpha \;=\; \mathtt{c}_1\tau'_{11}...\tau'_{1k_1} \mid ... \mid \mathtt{c}_m\tau'_{m1}...\tau'_{mk_m}).$$

For each $i = 1, ..., m$, given term relations $r_{ij} \in Rel(\tau_{ij}[\delta/\alpha], \tau'_{ij}[\delta'/\alpha])$ for $j = 1, ..., k_i$, we can form a term relation

$$\mathtt{c}_i r_{i1}...r_{ik_1} \;=\; \{(\mathtt{c}_i\overline{M_{k_i}}, \mathtt{c}_i\overline{M'_{k_i}}) \mid \forall j = 1, ..., k_i.\, (M_j, M'_j) \in r_{ij}\}.$$

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

Definition 10. *A relational action Δ comprises a family of mappings*

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

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 must satisfy the five conditions 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. *If δ is as in (1), then $\Delta_\delta(\overline{r_n}/\overline{\alpha_n})$ is a fixed point of the mapping*

$$r \;\mapsto\; \left(\bigcup_{i=1}^{n} \mathtt{c}_i\, (\Delta_{\tau_{i1}}(r/\alpha, \overline{r_n}/\overline{\alpha_n})) \;...\; (\Delta_{\tau_{ik_i}}(r/\alpha, \overline{r_n}/\overline{\alpha_n}))\right)^{\top\top}$$

5. *Assuming $ftv(\tau) \subseteq \{\overline{\alpha_n}, \overline{\alpha'_m}\}$ and $ftv(\overline{\tau'_m}) \subseteq \{\overline{\alpha_n}\}$,*

$$\Delta_{\tau[\overline{\tau'_m}/\overline{\alpha'_m}]}(\overline{r_n}/\overline{\alpha_n}) = \Delta_\tau(\overline{r_n}/\overline{\alpha_n}, (\Delta_{\overline{\tau'_m}}(\overline{r_n}/\overline{\alpha_n}))/\overline{\alpha'_m})$$

To see that the third clause above 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 9, we thus 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 Definition 10.

We now define the relational actions μ and ν. Our focus on contextual equivalence — which identifies programs as much as possible unless there are observable reasons for not doing so — will mean that we are concerned primarily with ν in this paper. But since the results below hold equally well for μ and ν, we follow Pitts' lead and state results in the neutral notation of an arbitrary relational action Δ.

Definition 11. *The relational action μ is given as in Definition 10, where the least fixed point is taken when defining the relational action at a data type δ in the fourth clause above. The relational action ν is defined similarly, except that the greatest, rather than the least, fixed point is taken in the fourth clause. The action μ gives an inductive character to the action at data types, while ν gives a coinductive character at data types.*

Taking $n = 0$ in Definition 10, 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 $\Delta_\tau() \in Rel(\tau, \tau)$. Pitts has shown that this relation coincides with the relation of contextual equivalence of closed PolyFix 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

$$\Gamma \vdash M =_{ctx} M' : \tau \quad \Leftrightarrow \quad \Gamma \vdash M \,\Delta\, M' : \tau. \tag{7}$$

The observation (7) guarantees that the logical relation Δ corresponds to the operational semantics of PolyFix. In particular, the definition of $\Delta_{\tau_1 \to \tau_2}$ in the second clause of Definition 10 reflects the fact that termination at function types is not observable in PolyFix. 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 PolyFix contextual equivalence.

As suggested by Pitts, it is possible to define call-by-value and call-by-name [11] versions of PolyFix. In each case, the definition of the relation $(-)\top(-)$ and the action of arrow types on term relations must be modified to reflect the appropriate operational semantics and notion of observability. Defining a call-by-name PolyFix also requires a slightly different notion of frame stack. The full development of these ideas for a call-by-value version of a subset of PolyFix is given in Pitts [12]; the details for a full call-by-value PolyFix and a call-by-name PolyFix 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 (7). Proposition 1 guarantees that Δ is reflexive.

Proposition 1. *If Δ is a relational action, then for each closed type τ and each closed term M, $(M, M) \in \Delta_\tau()$.*

Proposition 2. *For all closed types τ and closed terms M and M' of type τ,*

$$M =_{ctx} M' : \tau \;\Leftrightarrow\; \forall S \in Stack(\tau).\, S \top M \;\Leftrightarrow\; S \top M'$$

5.3 Proof of the Substitution Theorem

Proof of Theorem 1: Let Δ be a relational action and suppose the hypotheses of the Substitution Theorem hold. Since M and its type are closed, Proposition 1 ensures that

$$(M, M) \in \Delta_{\forall\alpha.(\tau_{11}\to\ldots\to\tau_{1k_1}\to\alpha)\to\ldots\to(\tau_{m1}\to\ldots\to\tau_{mk_m}\to\alpha)\to\alpha}() \tag{8}$$

Applying the definition of Δ for $\forall$-types shows that (8) holds iff for all closed types τ' and τ and for all $r \in Rel(\tau', \tau)$,

$$(M\tau', M\tau) \in \Delta_{(\tau_{11}\to\ldots\to\tau_{1k_1}\to\alpha)\to\ldots\to(\tau_{m1}\to\ldots\to\tau_{mk_m}\to\alpha)\to\alpha}(r^{\top\top}/\alpha)$$

An m-fold application of the definition of Δ for arrow types ensures that for all closed types τ' and τ, for all $r \in Rel(\tau', \tau)$, for all $i \in \{1, ..., m\}$, and for all pairs of closed terms $(\oplus'_i, \oplus_i) \in \Delta_{\tau_{i1}\to\ldots\to\tau_{ik_i}\to\alpha}(r^{\top\top}/\alpha)$, (8) holds iff $(M\tau'\overline{\oplus'_m}, M\tau\overline{\oplus_m}) \in \Delta_\alpha(r^{\top\top}/\alpha)$, *i.e.*, iff $(M\tau'\overline{\oplus'_m}, M\tau\overline{\oplus_m}) \in r^{\top\top}$. Expanding the condition on $(\oplus'_i, \oplus_i)$ for each $i = 1, ..., m$ shows it equivalent to the assertion that if $(a'_{ij}, a_{ij}) \in \Delta_{\tau_{ij}}(r^{\top\top}/\alpha)$ for each $j = 1, ..., k_i$, then $(\oplus'_i\overline{a'_{ik_i}}, \oplus_i\overline{a_{ik_i}}) \in r^{\top\top}$. Since (8) holds, we conclude that for all closed types τ' and τ and for all $r \in Rel(\tau', \tau)$,

$$\begin{array}{l}\text{if, for all } i = 1, ..., m,\\ \quad (a'_{ij}, a_{ij}) \in \Delta_{\tau_{ij}}(r^{\top\top}/\alpha) \text{ for all } j = 1, ..., k_i \text{ implies } (\oplus'_i\overline{a'_{ik_i}}, \oplus_i\overline{a_{ik_i}}) \in r^{\top\top},\\ \qquad \text{then } (M\tau'\overline{\oplus'_m}, M\tau\overline{\oplus_m}) \in r^{\top\top}\end{array} \tag{9}$$

Note that all of the terms appearing in (9) are closed.

Now consider the instantiation

$$\begin{array}{l}\tau' = \delta\\ r = \{(M, M') \mid \mathtt{cata}^\delta\,\tau\,\overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\;M\;=_{ctx}\;M' : \tau\}\\ \oplus'_i = \phi_i(\mathtt{c}_{u_1}, ..., \mathtt{c}_{u_p}, \mu_{v_1}, ..., \mu_{v_q})\\ \oplus_i = \phi_i(n_{u_1}, ..., n_{u_p}, \mu'_{v_1}, ..., \mu'_{v_q})\end{array}$$

If we can verify that the hypotheses of (9) hold, then we may conclude that

$$\begin{array}{l}\quad\mathtt{cata}^\delta\,\tau\,\overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\;\;(M\,\delta\,\overline{\phi_m(\mathtt{c}_{u_1}, ..., \mathtt{c}_{u_p}, \mu_{v_1}, ..., \mu_{v_q})})\\ =_{ctx}\;M\,\tau\,\overline{\phi_m(n_{u_1}, ..., n_{u_p}, \mu'_{v_1}, ..., \mu'_{v_q})} : \tau\end{array}$$

Then since $\mathtt{augment}^\delta\,M\,\overline{\mu_{v_q}} \;=_{ctx}\; M\,\delta\,\overline{\phi_m(c_{u_1}, ..., c_{u_p}, \mu_{v_1}, ..., \mu_{v_q})} : \delta$, we will have proved Theorem 1.

To verify that (9) holds, we first prove that r is $\top\top$-closed, and thus that r coincides with the PolyFix contextual equivalence relation. To see this let $(M, M') \in r^{\top\top}$. We show that $\mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1}, ..., n_{u_p}, n_{v_1}, ..., n_{v_q})}\;M\;=_{ctx} M' : \tau$. Let S be the "stack equivalent"

$$Id \circ \mathtt{case}\ -\ \mathtt{of}\ \{...\}$$

of the evaluation context $\mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}$. Then S is such that for all $N:\delta$,

$$S\,N \;=_{ctx}\; \mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\;N \;:\; \tau \tag{10}$$

since

$$\begin{array}{ll}
& \mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\;N \\
=_{ctx} & (\lambda d.\,\Lambda\alpha.\,\lambda\overline{f_m}.\mathtt{case}\;\;d\;\;\mathtt{of} \\
& \quad \{... \mid \mathtt{c}_i\overline{x_{k_i}} \Rightarrow \\
& \qquad f_i\phi_{ik_i}(\mathtt{cata}^\delta...\,x_{z_1}\alpha\overline{f_m},...,\mathtt{cata}^\delta...\,x_{z_p}\alpha\overline{f_m},x_{y_1},...,x_{y_q}) \mid ...\}) \\
& \qquad\quad N\;\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})} \\
=_{ctx} & \mathtt{case}\;\;N\;\;\mathtt{of}\;\;\{...\} \\
=_{ctx} & (Id \circ \mathtt{case}\;\;-\;\;\mathtt{of}\;\;\{...\})\;N \\
=_{ctx} & S\,N
\end{array}$$

The first equivalence is by (5) and the definition of $\mathtt{cata}$, the second is by repeated application of (2) and (3), 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 of frame stacks by

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

and

$$S' \mathbin{@} (S \circ F) = (S' \mathbin{@} S) \circ F$$

then

$$(S' \mathbin{@} S) \top M \;\Leftrightarrow\; S' \top (SM) \tag{11}$$

Moreover, for any $S' \in \mathit{Stack}(\tau)$, $(S' \mathbin{@} S,\, S')$ has the property that for all (N, N') with $\mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\;N =_{ctx} N' : \tau$,

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

The first equivalence by (11), and the second is by Proposition 2 and (10) and the fact that $=_{ctx}$ is transitive. Together with (9), the fact that $(M, M') \in r^{\top\top}$ implies that

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

But then

$$\begin{array}{ll}
S' \top M' \Leftrightarrow & (S' \mathbin{@} S) \top M \\
\Leftrightarrow & S' \top S\,M \\
\Leftrightarrow & S \top \mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\;M
\end{array}$$

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

$$\forall S' \in \mathit{Stack}(\tau).\; S' \top M \;\;\Leftrightarrow\;\; S' \top \mathtt{cata}^\delta\,\tau\;\overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\;M$$

By Proposition 2, we therefore have

$$M' =_{ctx} \mathtt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\ M\ :\ \tau$$

as desired.

To verify the hypotheses of (9), observe that since the type of M is closed, each τ_{ij} is either a closed type or is precisely α. In the first case, $\Delta_{\tau_{ij}}(r^{\top\top}/\alpha)$ is precisely $\Delta_{\tau_{ij}}()$. Thus, if $(a'_{ij}, a_{ij}) \in \Delta_{\tau_{ij}}(r^{\top\top}/\alpha)$, then by Proposition 1 then $a'_{ij} =_{ctx} a_{ij} : \tau_{ij}$. In the second case, we have $\Delta_{\tau_{ij}}(r^{\top\top}/\alpha) = r^{\top\top} = r$, *i.e.*, $\mathtt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\ a'_{ij} =_{ctx} a_{ij}\ :\ \tau$. Since $=_{ctx}$ is a congruence, equivalences (2) through (5) guarantee that

$$\mathtt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}(\oplus'_i\overline{a'_{ik_i}})\ =_{ctx}\ \ \oplus_i\overline{a_{ik_i}}\ :\ \tau$$

i.e., that $(\oplus'_i\overline{a'_{ik_i}}, \oplus_i\overline{a_{ik_i}}) \in r$. By (9) we conclude that $(M\tau'\overline{\oplus'_m}, M\tau\overline{\oplus_m}) \in r$, *i.e.*, that

$$\begin{aligned} &\mathtt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},n_{v_1},...,n_{v_q})}\ \ (\mathtt{augment}^{\delta}\ M\ \overline{\mu_{v_q}}) \\ =_{ctx}\ \ &M\ \tau\ \overline{\phi_m(n_{u_1},...,n_{u_p},\mu'_{v_1},...,\mu'_{v_q})}\ :\ \tau \end{aligned}$$

It is also possible to derive $\top\top$-closedness of r as a consequence of (the analogue for non-list algebraic data types of) Lemma 6.1 of Pitts [14], but in the interest of keeping this paper as self-contained as possible, we choose to prove it directly.

6 Conclusion

In this paper we have defined a generalization of `augment` for lists for every algebraic data type, and used Pitts' characterization of contextual equivalence for PolyFix to prove the correctness of the corresponding `cata-augment` fusion rules for polymorphic lambda calculi supporting fixed point recursion at the level of terms and recursion via data types with non-strict constructors at the level of types. More specifically, we have shown that programs in such calculi which have undergone generalized `cata-augment` fusion are contextually equivalent to their unfused counterparts. The correctness of short cut fusion for algebraic data types, as well as of `cata-augment` fusion for lists, are special cases of this result.

The construct `augment` can be interpreted as constructing substitution instances of algebraic data structures. The generalized `cata-augment` rule can be seen as a means of optimizing compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of them.

Acknowledgments. I am grateful to Olaf Chitil, Graham Hutton, and Andrew Pitts for helpful discussions on the topic of this paper. I also thank the volume

editor and the anonymous referees for their comments. 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

1. Abadi, M. $\top\top$-closed relations and admissibility. *Mathematical Structures in Computer Science* 10, pp. 313 – 320, 2000.
2. Bainbridge, E., Freyd, P., Scedrov, A., and Scott, P.J. Functorial polymorphism. *Theoretical Computer Science* 70, pp. 35 – 64, 1990. Corrigendum in *Theoretical Computer Science* 71, p. 431, 1990.
3. Chitil, O. Type inference builds a short cut to deforestation. In *Proceedings, International Conference on Functional Programming*, pp. 249 – 260, 1999.
4. Gill, A. *Cheap Deforetation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.
5. Gill, A., Launchbury, J., and Peyton Jones, S. L. A short cut to deforestation. In *Proceedings, Conference on Functional Languages and Computer Architecture*, pp. 223 – 232, 1993.
6. Hu, Z., Iwasaki, H., and Takeichi, M. Deriving structural hylomorphisms from recursive definitions. in *Proceedings, International Conference in Functional Programming*, pp. 73 – 82, 1996.
7. Johann, P. An implementation of warm fusion. Available at `ftp://ftp.cse.ogi.edu/pub/pacsoft/wf/`, 1997.
8. Johann, P. and Visser, E. Warm fusion in Stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artifical Intelligence* 29(1-4), pp. 1 – 34, 2000.
9. Németh, L. *Catamorphism Based Program Transformations for Non-strict Functional Languages*. Draft, PhD thesis, Glasgow University, 2000.
10. Onoue, Y., Hu, Z., Iwasaki, H., and Takeichi, M. A calculational system HYLO. In *Proceedings, IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pp. 76 – 106, 1997.
11. Plotkin, G. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science* 1, pp. 125 – 159, 1975.
12. Pitts, A. Existential types: Logical relations and operational equivalence. In *Proceedings, International Colloquium on Automata, Languages, and Programming*, LNCS vol. 1443, pp. 309 – 326, 1998.
13. Pitts, A. Parametric Polymorphism, Recursive Types, and Operational Equivalence. Unpublished Manuscript.
14. Pitts, A. Parametric Polymorphism and Operational Equivalence. *Mathematical Structures in Computer Science* 10, pp. 1 – 39, 2000.
15. Takano, A. and Meijer, E. Short cut deforetation in calculational form. In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 324 – 333, 1995.
16. Reynolds, J. C. Types, abstraction, and parametric polymorphism. *Information Processing* 83, pp. 513 – 523, 1983.
17. Sheard, T. and Fegaras, L. A fold for all seasons. In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 233 – 242, 1993.
18. Wadler, P. Theorems for Free! In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 347 – 359, 1989.