A Generalization Of Short-Cut Fusion And Its Correctness Proof

By: **Patricia Johann**

Abstract

Short-cut fusion is a program transformation technique that uses a single, local transformation—called the foldr-build 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. For each algebraic data type we then define a generalization of build which constructs substitution instances of its associated data structures, and use Pitts' techniques to prove the correctness of a contextual equivalence-preserving fusion rule which generalizes short-cut fusion. These rules optimize compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of those data structures.

Johann, Patricia (2002) "A Generalization Of Short-Cut Fusion And Its Correctness Proof". *Higher-Order and Symbolic Computation*, 15, 273–300 Version Of Record Available From www.springer.com

A Generalization Of Short-Cut Fusion And Its Correctness Proof

Patricia Johann

1. Introduction

Fusion [7, 8, 22, 23] is the process of removing certain intermediate data structures from modularly constructed functional programs. *Short-cut fusion* [6, 7] is a particular fusion technique that 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.

Although Gill, Launchbury, and Peyton Jones showed that short-cut fusion can successfully fuse a wide variety of list-processing functions [7], its applicability is limited because list-producing functions cannot always be usefully expressed in terms of `build`. This observation led Gill [6] 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 fusing compositions of list-processing functions.

Fusion techniques have typically been developed first for lists, and only later have their generalizations to more general algebraic data types been considered. The investigation of short-cut fusion has proceeded along precisely these lines, with short-cut fusion for lists giving rise to generalizations for non-list algebraic data types, as well as to the incorporation of these generalizations into a number of automatic fusion tools (e.g. [4, 6, 9, 11, 15, 16]). 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 such data type a `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 the resulting generalized `cata-augment` fusion rules as optimizing compositions of functions that uniformly consume algebraic data structures with functions that uniformly produce substitution instances of those data structures.

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 ones exhibiting order-of-magnitude efficiency increases over those from which they are derived [13]. Nevertheless, there remain difficulties associated with their use. One of the most substantial is that these fusion techniques are rarely proved correct.

Short-cut fusion and its generalizations are traditionally 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, and their application to functional programs has typically been justified by appealing either to intuition about the operational behavior of `cata`, `build`, and `augment`, or else to Wadler's "free theorems" [25]. 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 conditions are known to be satisfied by models supporting a relationally parametric structure. Bainbridge et al. [2] have shown parametric models to exist for some higher-order polymorphic languages, but because these models fail to model fixed point recursion they do not adequately accommodate short-cut fusion and its generalizations.

While it may be possible to extend the models of Bainbridge et al. 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. But novel operationally-based techniques now make it possible to construct parametric models for an interesting class of such languages [18, 19], and thus to prove the correctness of short-cut fusion and its generalizations for them.

1.2. Proving correctness

In this paper we prove the correctness of generalized `cata-augment` fusion for algebraic data types in a class of calculi supporting both higher-order polymorphic functions and fixed point recursion. Correctness 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 for these calculi are all immediate consequences of this result; in particular, the correctness of short-cut fusion, whose correctness proof this paper generalizes, is worked out in detail in [10]. But because functional languages typically support features that are not modeled in the calculi considered here, our results do not apply to them directly. Nevertheless, our results do make some progress toward proving the correctness of short-cut fusion and its generalizations 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 generalized `cata-augment` 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 [18, 19]. 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' relational characterization of contextual equivalence to demonstrate that programs in these calculi that have undergone generalized `cata-augment` fusion are contextually equivalent to their unfused counterparts. The semantic correctness of generalized `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 generalized `cata-augment` 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 in which types are quantified only at the outermost level—can be achieved by appropriately restricting the characterizations for the corresponding impredicative ones.

The remainder of this paper is organized as follows. Section 2 informally discusses short-cut fusion and `cata-augment` fusion for both lists and non-list algebraic data types. In Section 3 we introduce the polymorphic lambda calculus PolyFix for which we formalize and prove the correctness of generalized `cata-augment` fusion; 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 in PolyFix, 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 examples of such functions: `cata-Expr` and `eval` consume, and `subst` produces, terms in the simple expression language given by

```
Expr t = Op Ops t t | Var t | Lit Nat
Ops    = Add | Sub | Mul | Div
```

In Figure 1, `cata-Expr` denotes the standard catamorphism over `Expr`, and `app` takes as input an element of `Ops` and returns the corresponding operator on two natural numbers. In the informal discussion in this section we express program fragments in a Haskell-like

```
cata-Expr :: forall t. forall t'.
               (Ops -> t'-> t'-> t') -> (t -> t') ->
               (Nat -> t') -> Expr t -> t'

cata-Expr = /\t t'. \o v l e.
             case e of
               Op op w z -> o op (cata-Expr t t' o v l w)
                                 (cata-Expr t t' o v l z)
               Var x -> v x
               Lit i -> l i

eval :: forall t. Expr t -> Nat
eval = /\t. \e.
         case e of
           Op op w z -> app op (eval t w) (eval t z)
           Var x -> error
           Lit i -> i

subst :: forall t. (t -> Expr t) -> Expr t -> Expr t
subst = /\t. \env e. case e of
                       Op op w z -> Op op (subst t env w)
                                          (subst t env z)
                       Var x -> env x
                       Lit i -> Lit i
```

Figure 1. Recursive functions on expressions.

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`.

Using the functions in Figure 1 we can define, for example, the function `substEval` which evaluates the result of applying a substitution to an expression:

```
substEval :: forall t. (t -> Expr t) -> Expr t -> Nat
substEval = /\t. \env e. eval t (subst t env e)
```

Unfortunately, modularly constructed programs like `substEval` 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 `substEval`, for instance, is straightforward and modular, but it unnecessarily constructs and then traverses the intermediate substitution instance of the expression `e`. This requires processing the expression `e` twice. Even in lazy languages this is expensive, both slowing execution time and increasing heap 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 `substEval` is defined as

```
substEval' :: forall t. (t -> Expr t) -> Expr t -> Nat
substEval' = /\t. \env e.
               case e of
                  Op op w z -> app op (substEval' t env w)
                                      (substEval' t env z)
                  Var x -> eval t (env x)
                  Lit i -> i
```

No intermediate expression is generated by `substEval'`.

Experienced programmers writing a function to evaluate substitution instances of expressions would likely produce `substEval'` rather than `substEval`; small functions like `substEval` 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

One commonly employed technique for eliminating intermediate data structures from functional programs is short-cut fusion for lists [6, 7]. Short-cut fusion for lists uses the `foldr-build` rule to fuse compositions of list-processing functions via applications of traditional fold/unfold program transformation steps. In order to participate in short-cut fusion, list-consuming functions must be expressible in terms of `foldr` and list-producing functions must be expressible in terms of `build`.

The function `foldr` is just the standard catamorphism for lists. It is therefore easily generalized to a more general data structure-consuming construct called `cata` that can be instantiated to arbitrary algebraic data types. For each such data type `D` we denote the instantiation of `cata` to `D` by `cata-D`; in particular, we write `cata-List` for `foldr` in the remainder of this paper. We can similarly generalize the `build` function of Gill et al. to a more general data structure-producing construct `build` which can be instantiated to arbitrary algebraic data types; we denote by `build-D` the instantiation of `build` to `D` for each such data type `D`. Finally, with `build` and `cata` in hand we can define a `cata-build` rule which generalizes the `foldr-build` rule for lists to arbitrary algebraic data types. In the remainder of this subsection we describe, informally, this generalized `cata-build` rule and use functions over our simple expression data type to illustrate the generalized short-cut fusion technique to which it gives rise.

Operationally, `cata-D` takes as input types `t'`, `t1`, . . . ,`tn`, appropriately typed replacement functions for each of `D`'s constructors, and a data structure `d` of the data type `D t1`, . . . ,`tn`. It replaces all (fully applied) occurrences of `D`'s constructors in `d` by their corresponding replacement functions. The result is a value of type `t'`. For example, `cata-List` is given by

```
cata-List :: forall t. forall t'. (t -> t' -> t') -> t' ->
                                         List t -> t'
cata-List = /\t t'. \c n xs.
              case xs of
                 Nil -> n
                 Cons z zs -> c z (cata-List t t' c n zs)
```

and the definition of `cata-Expr` is as in Figure 1.

The function `build-D`, on the other hand, takes as input types `t1`, . . . ,`tn` and a term `M` providing a type-independent template for constructing "abstract" data structures from values of types `t1`, . . . ,`tn`. It instantiates all (fully applied) occurrences of the "abstract" constructors which appear in the "abstract" data structure specified by `M` with the corresponding "concrete" constructors of `D`. The result is a data structure of type `D t1`, . . . ,`tn`. For example, if `M` is any term with type

```
forall a. (Ops -> a -> a -> a) -> (t -> a) -> (Nat -> a) -> a
```

then

```
build-Expr t M = M (Expr t) Op Var Lit
```

and if `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 data structure-consuming and -producing functions defined using `cata-D` and `build-D` can be fused via a generalized `cata-build` rule. The instantiation of this rule to expressions ensures that

If `M` is a closed term of type

```
forall a. (Ops -> a -> a -> a) -> (t -> a)
                 -> (Nat -> a) -> a
```

then any occurrence of

```
cata-Expr t t' o v l (build-Expr t M)
```

in a program can be replaced by

```
M t' o v l
```

Similarly, the instantiation of this generalized `cata-build` rule to lists guarantees that

If `M` is a closed 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
```

Analogous rules can be used to fuse functions that produce and consume intermediate data structures of other algebraic data types.

The generalized `cata-build` rule and the generalized short-cut fusion technique to which it gives rise make sense intuitively: the result of a computation is the same regardless of whether a template `M` is first applied to (an instance of) a data type and its constructors and the constructors in the resulting data structure are then replaced by their corresponding constructor replacement functions, or the abstract constructors in (an appropriate instance of) `M` are replaced by their constructor replacement functions directly.

Figure 2 shows the `build-cata` forms of the functions in Figure 1. The fused function `substEval'` can be derived from `substEval` by inlining these definitions and applying short-cut fusion for expressions in conjunction with standard program simplifications.

2.2. *Cata-augment fusion*

Although generalized short-cut fusion can be used to fuse many compositions of data structure-processing functions, some compositions involving common functions are

```
eval :: forall t. Expr t -> Nat
eval = /\t. \e. cata-Expr t Nat app (\v -> error) id e

subst :: forall t. (t -> Expr t) -> Expr t -> Expr t
subst = /\t. \env e. build-Expr t
                       (/\a. \o v l. cata-Expr t a o
                          (cata-Expr t a o v l . env) l e)
```

Figure 2. Functions in build-cata form.

problematic. This is because the argument `M` to `build` must abstract *all* of the concrete constructors that appear in the data structure it produces—not just the "top-level" ones contributed by `M` itself.

To see why, suppose that we want to express the function `subst` for expressions over an arbitrary type `t` in terms of `build-Expr` and `cata-Expr`. It is tempting to write

```
subst = /\t. \env e. build-Expr t
                      (/\a. \o v l. cata-Expr t a o env l e)
```

but the expression on the right hand side is ill-typed: `env` is of type `t -> Expr t`, whereas `cata-Expr`'s replacement for `Var` needs to be of the more general type `t -> a`. The problem here is that the constructors in the expressions introduced by `env` are part of the result of `subst`, but they are not properly abstracted by `build-Expr`.

One solution is to use `cata-Expr` to prepare the constructors in the expressions introduced by `env` for abstraction via `build-Expr`. This entails consuming each such expression with `cata-Expr t a o v l`. The result is the `build-cata` form for `subst` which appears in Figure 2. Although this solution does indeed provide a replacement of type `t -> a` for `env`, it does so by introducing extra data structure consumptions into the computation. Unfortunately, subsequent removal of such consumptions via fusion cannot be guaranteed, even in the case of lists [6].

An alternative solution is to generalize `build-Expr` to abstract the expressions introduced by `env`. This yields a new construct `augment-Expr` defined by

```
augment-Expr t M mu_Var mu_Lit = M (Expr t) Op mu_Var mu_Lit
```

where `mu_Var :: t -> Expr t` and `mu_Lit :: Nat -> Expr t`. Constructing the substitution instance of an expression `e` relative to an environment `env` is easily and efficiently expressed in terms of `cata-Expr` and `augment-Expr`:

```
subst = /\t. \env e. augment-Expr t
                      (/\a. \o v l. cata-Expr t a o v l e)
                      env
                      Lit
```

Moreover, this definition of `augment-Expr` gives

```
build-Expr t M = augment-Expr t M Op Var Lit
```

Gill et al. [7] used the analogous observation for lists to define their original augment. They also defined a corresponding cata-augment rule for fusing compositions of functions written in terms of cata-List and augment-List. Its analogue for expressions is:

Let t be a type, let

```
mu_Var :: t -> Expr t
```

and let

```
mu_Lit :: Nat -> Expr t
```

In addition, let

```
M :: forall a. (Ops -> a -> a -> a) ->
         (t -> a) -> (Nat -> a) -> a
```

be a closed term. Then any occurrence of

```
cata-Expr t t' o v l
          (augment-Expr t M mu_Var mu_Lit)
```

in a program can be replaced by

```
M t' o (cata-Expr t t' o v l . mu_Var)
       (cata-Expr t t' o v l . mu_Lit)
```

Applying this rule to the augment and cata forms of subst and eval, respectively, produces substeval' even while avoiding the additional data structure consumptions required by build. A similar rule can be derived for every algebraic data type.

For every algebraic data type D, the generalized cata-build rule for D is just the special case of the generalized cata-augment rule for D in which D's constructor replacement functions have been specialized to their corresponding constructors, augment-D has been replaced by build-D, and the arguments to M involving cata-D have all been simplified. Although the generalized cata-augment rule does not eliminate the entire intermediate data structure produced by augment, it does avoid production and subsequent consumption of the part of the structure contributed by M.

3. PolyFix and contextual equivalence

In this section we introduce Pitts' PolyFix, the polymorphic lambda calculus for which we formalize, and prove the correctness of, generalized cata-augment fusion. We also outline those aspects of contextual equivalence for PolyFix terms which are needed in this endeavor.

Our presentation is heavily influenced by [19] and [18]. The latter is an unpublished manuscript containing a partially complete development of contextual equivalence for calculi supporting non-list algebraic data types. Since this development is entirely analogous to that in [19] for calculi supporting only lists, and since precisely the same techniques are used to investigate contextual equivalence in both settings, we refer the reader to [19], rather than [18], for proof details at several places below.

3.1. PolyFix: The fixed point calculus

The *Polymorphic Fixed Point Calculus* PolyFix combines the Girard-Reynolds polymorphic lambda calculus with fixed point recursion à la Plotkin's FPC calculus at the level of terms and (positive) recursion via non-strict constructors at the level of types [5, 20]. 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. There, the Haskell-like syntax

$$\mathtt{data}\big(\alpha = c_1 \tau_{11}, \ldots, \tau_{1k_1} \mid \ldots \mid c_m \tau_{m1}, \ldots, \tau_{mk_m}\big) \tag{1}$$

is used for recursive data types, and provides an anonymous notation for a data type δ satisfying the fixed point equation

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

The injections into the m-fold sum are named explicitly by δ's constructors $\mathtt{c}_1, \ldots, \mathtt{c}_m$. Terms of type δ are introduced using δ's constructors and eliminated using case expressions. We write $\mathtt{c}_i^\delta$ to emphasize that the constructor $\mathtt{c}_i$ is associated with the data type δ. The types τ_{ij}, for $i = 1, \ldots, m$ and $j = 1, \ldots, k_i$, 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}. The notion of a type variable occurring positively in another type is defined in Definition 1 below. As the definition of *Bool* in the next example shows, recursive data types can be recursive in the trivial sense.

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	$::= \mathtt{data}(\alpha = \mathtt{c}_1\overline{\tau_{k_1}} \mid \ldots \mid \mathtt{c}_m\overline{\tau_{k_m}})$	
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
		$\mid\ \mathtt{c}_i^\delta\overline{M_{k_i}}$	data value
		$\mid\ \mathtt{case}\ M\,\mathtt{of}\ \{\mathtt{c}_1\overline{x_{k_1}} \Rightarrow M \mid \ldots \mid \mathtt{c}_m\overline{x_{k_m}} \Rightarrow M\}$	case expression

Figure 3. Syntax of PolyFix.

Example 1. The following are PolyFix data types:

$$\texttt{data}(\alpha = \texttt{True} \mid \texttt{False})$$
$$\texttt{data}(\alpha = \texttt{Succ}\,\alpha \mid \texttt{Zero})$$
$$\texttt{data}(\alpha = \texttt{Cons}\,\tau\,\alpha \mid \texttt{Nil})$$

We denote these types by *Bool*, *Nat*, and *List* τ, respectively.

Note that, in addition to being anonymous, PolyFix data types can also be parameterized and nested. In practice it may be convenient to restrict attention to finite sets of named, mutually recursive data types which are defined at top level, and whose names make their parameters explicit.

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 $M s_1 \ldots s_n$. Similarly, we write $\lambda\overline{x_n} : \overline{\tau_n}.\, M$ to indicate the n-fold abstraction $\lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n.\, M$.

The constructions $\forall\alpha(-)$, $\texttt{data}(\alpha = -)$, $\texttt{case}\ M\ \texttt{of}\ \{\ldots \mid c_i\overline{x_{k_i}} \Rightarrow M_i \mid \ldots\}$, $\lambda x : \tau.-$, and $\Lambda\alpha.-$ are binders, and free occurrences of the variables $x_1, \ldots, x_{k_i}$ become bound in the case expression

$$\texttt{case}\ D\ \texttt{of}\ \left\{\ldots \,\middle|\, c_i\overline{x_{k_i}} \Rightarrow M_i \,\middle|\, \ldots\right\}$$

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 $M[M'/x]$.

In order to be well-formed we require a data type as in (1) to have distinct data constructors c_i, $i = 1, \ldots, m$, and to be algebraic in the sense of the next definition.

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

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

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 $i = 1, \ldots, m$ and $j = 1, \ldots, k_i$.

$$\Gamma,\, x : \tau \vdash x : \tau \qquad\qquad \frac{\Gamma \vdash M : \tau \to \tau}{\Gamma \vdash \mathtt{fix}\, M : \tau}$$

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

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

$$\frac{\Gamma \vdash M_j : \tau_j[\delta/\alpha] \qquad j = 1, .., k_i}{\Gamma \vdash \mathtt{c}_i^{\delta} M_1 ... M_{k_i} : \delta} \text{ if } \delta \text{ is } \mathtt{data}(\alpha = ... \mid \mathtt{c}_i \overline{\tau_{ik_i}} \mid ...)$$

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

Figure 4. PolyFix type assignment.

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. 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. We also write $\Gamma, x : \tau$ for the typing environment obtained from $\Gamma = A,\ \Delta$ by extending the function Δ to map $x \notin dom(\Delta)$ to τ, and $\Gamma,\ \alpha$ for the extension of A with a type variable $\alpha \notin A$. Implicit in the notation $\Gamma \vdash M : \tau$ are four assumptions, namely that $\Gamma = A,\ \Delta$, that $ftv(M) \subseteq A$, that $ftv(\tau) \subseteq A$, and that $fv(M) \subseteq dom(\Delta)$. Note that if $\Gamma = A, \Delta$ and $\Gamma,\ \alpha \vdash M : \tau$ for some M and τ, then it follows from $\alpha \notin A$ that α does not appear among the free type variables of Γ. Our assumptions thus render unnecessary the side condition $\alpha \notin ftv(\Gamma)$ that usually accompanies the rule in Figure 4 for deriving type assignments of the form $\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau$.

The explicit type annotations on lambda-bound term variables and on constructors in data values 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 will 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 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 $\{1, \ldots, m\}$ for which $\alpha \in fv(\tau_{ij})$ for some $j \in \{1, \ldots, k_i\}$, and let $NonRec_\delta$ be the set $\{1, \ldots, m\} - Rec_\delta$. 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 $j \in \{1, \ldots, k_i\}$ for which $\alpha \in fv(\tau_{ij})$, and let $NonRecPos_{\mathtt{c}_i}$ be the set $\{1, \ldots, k_i\} - RecPos_{\mathtt{c}_i}$. We say that the indices in $RecPos_{\mathtt{c}_i}$ indicate the *recursive positions* of $\mathtt{c}_i$ and that the indices in $\mathit{NonRecPos}_{\mathtt{c}_i}$ 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 notational conventions introduced in the next definition allow us to order, and project onto the resulting sequence of, arguments to function abstractions. We use them to express `cata`, `build`, and `augment` in PolyFix.

Definition 2. Suppose δ is as in (1), τ is a closed type, $\mathit{Rec}_\delta = \{u_1, \ldots, u_p\}$, and $\mathit{NonRec}_\delta = \{v_1, \ldots, v_q\}$. Then for all $i = 1, \ldots, m$ and $\rho_i : \tau_{i1}[\tau/\alpha] \to \cdots \to \tau_{ik_i}[\tau/\alpha] \to \tau$, define

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

Further, for each $i = 1, \ldots, m$, if $\mathit{RecPos}_{\mathtt{c}_i} = \{z_1, \ldots, z_p\}$ and $\mathit{NonRecPos}_{\mathtt{c}_i} = \{y_1, \ldots, y_q\}$, then define

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

Finally, for each data type δ as in (1) define a corresponding polymorphic type τ_δ by

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

Using these conventions, we have

Definition 3. For each data type δ define

$\mathtt{build}^\delta \;:\; \tau_\delta \to \delta$
$\mathtt{build}^\delta = \lambda M : \tau_\delta.\, M\, \delta\, \overline{c_m}$
where for each $\mathtt{c}_i$, we define $c_i = \lambda \overline{p_{k_i}} : \overline{\tau_{k_i}[\delta/\alpha]}.\, \mathtt{c}_i\, \overline{p_{k_i}}$

$\mathtt{unbuild}^\delta \;:\; \delta \to \tau_\delta$
$\mathtt{unbuild}^\delta = \mathtt{fix}\big(\lambda h : \delta \to \tau_\delta.\, \lambda d : \delta.\, \Lambda\alpha.\, \lambda \overline{f_m} : \overline{\tau_{m1} \to \cdots \to \tau_{mk_m} \to \alpha}.$
$\quad\mathtt{case}\ d\ \mathtt{of}$
$\quad\quad\big\{ \ldots \,\big|\, \mathtt{c}_i \overline{x_{k_i}} \Rightarrow f_i \overline{\phi_{ik_i}\big(hx_{z_1}\alpha\overline{f_m}, \ldots hx_{z_p}\alpha\overline{f_m}, x_{y_1}, \ldots, x_{y_q}\big)} \,\big|\, \ldots \big\}\big)$
where $\mathit{RecPos}_{\mathtt{c}_i} = \{z_1, \ldots, z_p\}$ and $\mathit{NonRecPos}_{\mathtt{c}_i} = \{y_1, \ldots, y_q\}$

$\mathtt{cata}^\delta \;:\; \forall\alpha.\, \big(\tau_{11} \to \cdots \to \tau_{1k_1} \to \alpha\big) \to \cdots \to \big(\tau_{m1} \to \cdots \to \tau_{mk_m} \to \alpha\big) \to \delta \to \alpha$
$\mathtt{cata}^\delta = \Lambda\alpha.\, \lambda\overline{f_m}.\, \lambda d.\, \mathtt{unbuild}^\delta\; d\; \alpha\; \overline{f_m}$

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 `cata-List` $\tau = \mathtt{cata}^{\mathit{List}\,\tau}$ and `build-List` $\tau = \mathtt{build}^{\mathit{List}\,\tau}$. Note that constructors must be fully applied in well-formed PolyFix terms.

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

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

$$\frac{D \Downarrow \mathsf{c}_i^\delta \overline{M_{k_i}} \qquad M[\overline{M_{k_i}}/\overline{x_{k_i}}] \Downarrow V}{\mathtt{case}\ D\ \mathtt{of}\ \{\ldots \mid \mathsf{c}_i \overline{x_{k_i}} \Rightarrow M \mid \ldots\} \Downarrow V} \quad \text{if } \delta \text{ is } \mathtt{data}(\alpha = \ldots \mid \mathsf{c}_i \overline{\tau_{ik_i}} \mid \ldots)$$

Figure 5. PolyFix evaluation relation.

3.2. Operational semantics

The operational semantics of PolyFix is given by the *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 PolyFix *values* is given by

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

Note that function application is given a call-by-name semantics, constructors are non-strict, and type applications are not evaluated "under the Λ." Although PolyFix evaluation is deterministic, 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.

Recall that ground types have been replaced by data types in PolyFix. 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 take 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 of the value, if any, to which it evaluates. Although it may seem more natural to observe as much as one can about the results of evaluation, observing the entire data values, if any, to which programs evaluate can lead to too high a degree of intensionality. On the other hand, by considering programs in suitable contexts we can show that observing only the outermost constructors of the values, if any, to which programs evaluate leads to the same notion of contextual equivalence as merely observing whether or not they terminate. We use these observations to formalize contextual equivalence.

Writing $M \Downarrow$ to mean that $M \Downarrow V$ for some value V, we say that two PolyFix terms M_1 and M_2 such that $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$ are *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

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

That is, two PolyFix terms are contextually equivalent if they exhibit the same termination behavior in context. 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. We write $\Gamma \vdash M_1 =_{ctx} M_2 : \tau$ to indicate that M_1 and M_2 are contextually equivalent and have type τ with respect to Γ. If M_1 and M_2 are closed terms and τ is a closed type, then 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.*

4. A generalized cata-augment rule

In this section we state our main result, the correctness of generalized `cata-augment` fusion. This theorem allows us to generalize fusion via 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 theorem appears in Section 5.3.

We will consider contextual equivalence of only closed terms of closed type in the remainder of this paper. Contextual equivalence for open terms is reducible to contextual equivalence for closed terms of closed type [19].

Theorem 1. *Let δ be a closed data type as in* (1), *let τ_δ, $u_1, \ldots, u_p$, $v_1, \ldots, v_q$, and $\phi_1, \ldots, \phi_m$ be as in Definition* 2, *and let $M \in \mathit{Term}(\tau_\delta)$. In addition, let τ be a closed type and, for $i = 1, \ldots, m$ and $j = 1, \ldots, k_i$, define $\tau'_{ij} = \tau_{ij}[\tau/\alpha]$ and $\tau''_{ij} = \tau_{ij}[\delta/\alpha]$. Finally, for $i = 1, \ldots, m$ and $v \in \mathit{NonRec}_\delta$, let*

$$\begin{aligned} c_i &= \lambda \overline{p_{k_i}} : \overline{\tau''_{k_i}}.\mathtt{c}_i\, \overline{p_{k_i}}, \\ n_i &: \tau'_{i1} \to \cdots \to \tau'_{ik_i} \to \tau, \end{aligned}$$

and

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

be closed terms. Then

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

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

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

The functions μ_v for $v \in \mathit{NonRec}_\delta$ can be thought of as substitutions mapping appropriate combinations of arguments of types τ''_{vj}, $j = 1, \ldots, k_v$, to terms of type δ. They determine the portions of the intermediate data structure not contributed by M itself, i.e., the non-initial segments of the intermediate data structure. Theorem 1 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 Theorem 1 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 $\mathtt{cata}^\delta\ \overline{\phi_m(n_{u_1}, \ldots, n_{u_p}, n_{v_1}, \ldots, n_{v_q})}\ (\mu_v x_1 \ldots x_{k_v})$ exactly once.

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

$$\begin{aligned}\mathtt{augment}^\delta \ &:\ \tau_\delta \to \left(\tau''_{v_1 1} \to \cdots \to \tau''_{v_1 k_{v_1}} \to \delta\right) \to \cdots \to \\ &\qquad\quad \left(\tau''_{v_q 1} \to \cdots \to \tau''_{v_q k_{v_q}} \to \delta\right) \to \delta \\ \mathtt{augment}^\delta &= \lambda M.\, \lambda \overline{\mu_{v_q}}.\, M\, \delta\, \overline{\phi_m\left(c_{u_1}, \ldots, c_{u_p}, \mu_{v_1}, \ldots, \mu_{v_q}\right)}\end{aligned}$$

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

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

Writing $\mathtt{cata}^{\mathit{Expr}\,\tau}$ and $\mathtt{augment}^{\mathit{Expr}\,\tau}$ for the constructs `cata-Expr` τ and `augment-Expr` τ, respectively, the `cata-augment` rule for expressions given informally in Section 2 is easily formalized as an instance of the above contextual equivalence. Note that the generalized `cata-augment` rule allows the replacement terms for the nonrecursive data constructors to be specified by any appropriately typed substitutions $\mu_{v_1}, \ldots, \mu_{v_q}$. In this notation, the `cata-augment` rule for expressions requires two such substitutions, one corresponding to `Var` and one corresponding to `Lit`. The `cata-augment` rule for lists, on the other hand, requires only a substitution corresponding to `Nil`. Since `Nil` takes no term arguments, the substitution corresponding to it is usually denoted by the single list in the substitution's image.

For any data type δ, specializing μ_v to c_v for $v \in \mathit{NonRec}_\delta$ in $\mathtt{augment}^\delta\ M\ \overline{\mu_{v_q}}$ gives $\mathtt{build}^\delta\ M$, just as for lists. With this specialization, Theorem 1 yields the usual `cata-build` rule for algebraic data types, and so makes precise the sense in which the fusion technique to which it gives rise 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 cannot be applied to them, in all situations in which the rule is used to fuse programs the free variables in the term arguments to `augment` will already have been instantiated with closed terms.

5. Correctness of cata-augment fusion

To prove Theorem 1 we will define a logical relation which coincides with PolyFix 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 PolyFix contextual equivalence would enforce contextual equivalence of related terms. This would in turn incorporate into the theory of PolyFix contextual equivalence a notion of relational parametricity analogous to that introduced by Reynolds for the pure polymorphic lambda calculus [21].

Unfortunately, a naive approach to defining such a logical relation—i.e., an approach 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 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.) Pitts' notion of $\top\top$-closure, defined below, provides a criterion sufficient to guarantee this kind of admissibility [1].

The notion of $\top\top$-closure is defined in terms of the *structural termination relation* $\top$ on PolyFix terms. It 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. As shown by Pitts, analysis of evaluation contexts is aided by recasting them in terms of the notion of frame stack given in Definition 4 below; indeed, this frame stack realization of evaluation contexts gives rise to a structural characterization of termination of PolyFix program evaluation. The resulting structural termination relation $\top$ for PolyFix provides the key to appropriately specifying the clause for $\forall$-types in the logical relation which coincides with contextual equivalence.

The importance of the relation $\top$ is underscored by the observation that it is difficult to employ standard proof-theoretic techniques to establish termination properties—and therefore contextual equivalence—of PolyFix programs in the absence of a structurally defined relation which coincides with PolyFix termination. The termination relation derived directly from the rules in Figure 5 does not meet this criterion. In particular, the rule

$$\frac{F \Downarrow \lambda x : \tau.\, M \qquad M[A/x] \Downarrow}{FA \Downarrow}$$

for termination of function abstractions is not structural, since $M[A/x]$ need not appear as a subterm of *FA*. This can hinder efforts to prove that evaluation of a PolyFix term terminates by inducting on its structure, or to establish that two terms are contextually equivalent by, for instance, inducting on the proof that evaluation of one of them is terminating to construct a proof that evaluation of the other is as well. By contrast, the structurally defined relation $\top$, which captures termination of PolyFix terms, provides a much-needed tool for analyzing of PolyFix termination.

After introducing the structural termination relation $\top$ and sketching Pitts' characterization of contextual equivalence in terms of a logical relation based on the notion of $\top\top$-closure in Sections 5.1 and 5.2, we use the latter to prove Theorem 1 in Section 5.3.

5.1. $\top\top$-*Closed relations*

Definition 4. The grammar for PolyFix *frame stacks* is

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

where F ranges over *frames*:

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

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

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

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

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

Figure 6. Frame stack type judgements.

write

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

The operation $S, M \mapsto SM$ of *applying a frame 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:

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

Here, $F[M]$ is the term that results from replacing '$-$' by M in the frame F. Recalling from Section 3.1 that $Term(\tau)$ is the set of closed PolyFix terms M for which $\emptyset, \emptyset \vdash M : \tau$, we have that if $S \in Stack(\tau, \tau')$ and $M \in Term(\tau)$, then $SM \in Term(\tau')$. Unlike PolyFix evaluation, frame stack application is strict in its second argument. This follows from the fact that

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

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 structural termination relation $(-)\top(-)$ defined in Figure 7. More precisely, for all closed types τ, all closed data types δ, all frame stacks $S \in Stack(\tau, \delta)$, and all $M \in Term(\tau)$,

$$SM \Downarrow \text{ iff } S \top M$$

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. Similar unwinding theorems are used in several places in the literature; see, e.g. [3, 12, 14].

Definition 5. 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:

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

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

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

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

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

Figure 7. PolyFix structural termination relation.

Definition 6. 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 7. 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

$$\begin{array}{l} \text{for each pair } (S, S') \text{ of (appropriately typed) stacks,} \\ \quad \text{if for all } (N, N') \in r.\ S \top N \ \Leftrightarrow\ S' \top N' \\ \quad \text{then } S \top M \ \Leftrightarrow\ S' \top M' \end{array} \qquad (2)$$

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 8. *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 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 data constructors on term relations: Let δ and δ' be the closed data types

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

and

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

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

$$\mathtt{c}_i^{\delta} r_{i1} \ldots r_{ik_1} = \big\{\big(\mathtt{c}_i^{\delta}\overline{M_{k_i}}, \mathtt{c}_i^{\delta}\overline{M_{k_i}'}\big) \;\big|\; \text{for all } j = 1, \ldots, k_i .\, (M_j, M_j') \in r_{ij}\big\}.$$

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

Definition 9. A *relational action* Δ comprises a family of mappings

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

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 following five conditions:

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

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

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

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

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 $r \in \mathit{Rel}(\sigma, \sigma')$ for closed types σ and σ' to the relation $\Delta_\tau(r^{\top\top}/\alpha, \overline{r_n}/\overline{\alpha_n}) \in \mathit{Rel}(\tau[\overline{\tau_n}/\overline{\alpha_n}][\sigma/\alpha], \tau[\overline{\tau'_n}/\overline{\alpha_n}][\sigma'/\alpha])$. According to Definition 8, we therefore have $\forall r.\ \Delta_\tau(r^{\top\top}/\alpha, \overline{r_n}/\overline{\alpha_n}) \in \mathit{Rel}(\forall\alpha.\tau[\overline{\tau_n}/\overline{\alpha_n}], \forall\alpha.\tau[\overline{\tau'_n}/\overline{\alpha_n}])$, as required by Definition 9.

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 state results in the neutral notation of an arbitrary relational action Δ.

Definition 10. The relational action μ is given as in Definition 9, 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 least and greatest fixed points of the mapping in the fourth clause of Definition 9 exist by Tarski's fixed point theorem [24]: each of the sets $\mathit{Rel}(\tau, \tau')$ forms a complete lattice with respect to set inclusion, and the restriction to algebraic data types ensures that the mapping is monotone. 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 9, 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 $\Delta_\tau() \in \mathit{Rel}(\tau, \tau)$. Pitts has shown that this relation coincides with the relation of contextual equivalence of closed PolyFix terms at the closed type τ. In fact, he 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

$$\Gamma \vdash M =_{ctx} M' : \tau \quad \Leftrightarrow \quad \Gamma \vdash M \,\Delta\, M' : \tau \tag{3}$$

The observation (3) 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 9 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 a call-by-value PolyFix and a "lazy" PolyFix, i.e., a PolyFix with call-by-name evaluation in which termination at function types is observable. 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. In addition, defining a call-by-value 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 appears in [17]; the details for a full call-by-value PolyFix and a "lazy" 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` entails 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 three corollaries of (3).

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

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

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

Proposition 3. *Let τ be a closed type, let $M, M' \in \mathit{Term}(\tau)$, and write $M =_{kl} M' : \tau$ to indicate that, for all values V, $M \Downarrow V$ iff $M' \Downarrow V$. Then $M =_{kl} M' : \tau$ implies $M =_{ctx} M' : \tau$.*

The term relation $(-) =_{kl} (-) : \tau$ is called *Kleene equivalence* (at type τ). Together with the observation, immediate from Figure 5, that each of the equivalences (4) through (7) is a Kleene equivalence, Proposition 3 guarantees that for all terms M and M' of type τ_1 and A of type τ_2,

$$(\lambda x : \tau_2.\, M)A =_{ctx} M[A/x] \;:\; \tau_1 \tag{4}$$

$$(\Lambda\alpha.\, M)\tau_2 =_{ctx} M[\tau_2/\alpha] \;:\; \tau_1[\tau_2/\alpha] \tag{5}$$

$$\texttt{case}\ \texttt{c}_i\overline{M_{k_i}}\texttt{of}\ \left\{\ldots \,\middle|\, \texttt{c}_i\overline{x_{k_i}} \Rightarrow M' \,\middle|\, \ldots\right\} =_{ctx} M'\left[\overline{M_{k_i}}/\overline{x_{k_i}}\right] \;:\; \tau_1 \tag{6}$$

$$\texttt{fix}\, M =_{ctx} M(\texttt{fix}\, M) \;:\; \tau_1 \tag{7}$$

We are now prepared to establish our main result.

5.3. *Proof of Theorem 1*

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

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

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

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

An m-fold application of the definition of Δ for arrow types thus ensures that for all closed types τ' and τ, for all $r \in Rel(\tau', \tau)$, for all $i \in \{1, \ldots, m\}$, and all pairs of closed terms $(\oplus'_i, \oplus_i) \in \Delta_{\tau_{i1}\to\cdots\to\tau_{ik_i}\to\alpha}(r^{\top\top}/\alpha)$, (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 $i = 1, \ldots, m$ shows it equivalent to the assertion that if $(a'_{ij}, a_{ij}) \in \Delta_{\tau_{ij}}(r^{\top\top}/\alpha)$ for each $j = 1, \ldots, k_i$, 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 $r \in Rel(\tau', \tau)$,

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

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

Now consider the instantiation

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

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

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

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

To verify that (9) holds, we first prove that r is $\top\top$-closed. To see this, suppose that $(M, M') \in r^{\top\top}$. Our goal is to establish that

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

i.e., that (M, M') is in r. Let S be the "stack equivalent"

$$Id \circ \mathtt{case}\ -\ \mathtt{of}\,\{\cdots\}$$

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

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

since

$$\begin{aligned}
&\mathtt{cata}^{\delta}\,\tau\ \overline{\phi_m\left(n_{u_1}, \ldots, n_{u_p}, n_{v_1}, \ldots, n_{v_q}\right)}\ N\\
&\quad =_{ctx} \Big(\lambda d.\,\Lambda\alpha.\,\lambda\overline{f_m}.\mathtt{case}\ \ d\ \mathtt{of}\\
&\qquad\qquad \Big\{\ldots\ \Big|\ \mathtt{c}_i^{\delta}\overline{x_{k_i}} \Rightarrow f_i\overline{\phi_{ik_i}\left(\mathtt{cata}^{\delta}\ldots\, x_{z_1}\alpha\overline{f_m}, \ldots, \mathtt{cata}^{\delta}\ldots x_{z_p}\alpha\overline{f_m}, x_{y_1}, \ldots, x_{y_q}\right)}\ \Big|\ \ldots\Big\}\Big)\\
&\qquad\qquad N\ \tau\ \overline{\phi_m\left(n_{u_1}, \ldots, n_{u_p}, n_{v_1}, \ldots, n_{v_q}\right)}\\
&\quad =_{ctx} \mathtt{case}\ \ N\mathtt{of}\ \ \{\cdots\}\\
&\quad =_{ctx} (Id\ \circ\ \mathtt{case}\ \ -\mathtt{of}\ \ \{\cdots\})\,N\\
&\quad =_{ctx} SN
\end{aligned}$$

The first equivalence is by (7) and the definition of `cata`, the second is by repeated application of (4) and (5), 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

$$S\ @\ Id = S$$

and

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

then

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

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

$$(S'\ @\ S) \top N\ \ \Leftrightarrow\ \ S' \top SN\ \ \Leftrightarrow\ \ S' \top N'$$

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

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

But then

$$\begin{aligned} S' \top \texttt{cata}^{\delta}\,\tau\, \overline{\phi_m\big(n_{u_1},\ldots,n_{u_p},n_{v_1},\ldots,n_{v_q}\big)}\, M &\Leftrightarrow S' \top S M \\ &\Leftrightarrow (S' \mathbin{@} S) \top M \\ &\Leftrightarrow S' \top M' \end{aligned}$$

Here, the first equivalence is by (10) and Proposition 2, the second is by (11), and the third is by (12). Since S' was arbitrary we have shown that

for all $S' \in \mathit{Stack}(\tau)$.

$$S' \top \texttt{cata}^{\delta}\ \tau\ \overline{\phi_m\big(n_{u_1},\ldots,n_{u_p},n_{v_1},\ldots,n_{v_q}\big)}\ M \ \Leftrightarrow\ S' \top M'$$

By Proposition 2, we therefore have

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

i.e., $(M, M') \in r$, 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 $\Delta_{\tau_{ij}}(r^{\top\top}/\alpha) = r^{\top\top} = r$, so $\texttt{cata}^{\delta}\ \tau\ \overline{\phi_m(n_{u_1},\ldots,n_{u_p},n_{v_1},\ldots,n_{v_q})}\ a'_{ij} =_{ctx} a_{ij}\ :\ \tau$. Since $=_{ctx}$ is a congruence, equivalences (4) through (7) guarantee that

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

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

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

as desired.

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 [19], 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 those data structures.

Acknowledgments

I am grateful to Olaf Chitil, Graham Hutton, and Andrew Pitts for fruitful discussions on the topic of this paper. I also thank the anonymous referees for their helpful 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** (2000) 313–320.
2. Bainbridge, E.S., Freyd, P.J., Scedrov, A., and Scott, P.J. Functorial polymorphism. *Theoretical Computer Science*, **70**(1) (1990) 35–64. Corrigendum in **71**(3) (1990) 431.
3. Bierman, G.M., Pitts, A.M., and Russo C.V. Operational semantics of Lily, a polymorphic linear lambda calculus with recursion. *Electronic Notes in Theoretical Computer Science*, **41**(3) (2000).
4. Chitil, O. Type inference builds a short cut to deforestation. In *Proceedings, International Conference on Functional Programming*, 1999, pp. 249–260.
5. Fiore, M. and Plotkin, G. An axiomatization of computationally adequate domain theoretic models of FPC. In *Proceedings, 9th Annual Symposium on Logic in Computer Science*, 1994, pp. 92–102.
6. Gill, A. *Cheap Deforestation for Non-strict Functional Languages*. PhD Thesis, Glasgow University, 1996.
7. Gill, A., Launchbury, J., and Peyton Jones, S.L. A short cut to deforestation. In *Proceedings, Conference on Functional Languages and Computer Architecture*, 1993, pp. 223–232.
8. Hu, Z., Iwasaki, H., and Takeichi, M. Deriving structural hylomorphisms from recursive definitions. In *Proceedings, International Conference on Functional Programming*, 1996, pp. 73–82.
9. Johann, P. An implementation of warm fusion. Available at `ftp://ftp.cse.ogi.edu/pub/pacsoft/wf/`, 1997.
10. Johann, P. Short cut fusion is correct. *Journal of Functional Programming*. To appear.
11. Johann, P. and Visser, E. Warm fusion in Stratego: A case study in generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, **29**(1–4) (2000) 1–34.
12. Lassen, S.B. *Relational Reasoning about Functions and Nondeterminism*. PhD Thesis, University of Aarhus, 1998.
13. Launchbury, J. and Sheard, T. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings, Conference on Functional Programming and Computer Architecture*, 1995, pp. 314–323.
14. Mason, I.A., Smith, S.F., and Talcott, C.L. From operational semantics to domain theory. *Information and Computation*, **128**(1) (1996) 26–47.
15. Németh, L. *Catamorphism Based Program Transformations for Non-Strict Functional Languages*. PhD Thesis, Glasgow University, 2000.
16. 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*, 1997, pp. 76–106.

17. Pitts, A. Existential types: Logical relations and operational equivalence. In *Proceedings, International Colloquium on Automata, Languages, and Programming*, LNCS, Vol. 1443, 1998, pp. 309–326.
18. Pitts, A. Parametric polymorphism, recursive types, and operational equivalence. Unpublished Manuscript (1998).
19. Pitts, A. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, **10** (2000) 1–39.
20. Reynolds, J.C. Toward a theory of type structure. Paris Colloquium on Programming, LNCS, Vol. 19, 1974, pp. 408–425.
21. Reynolds, J.C. Types, abstraction, and parametric polymorphism. *Information Processing*, **83** (1983) 513–523.
22. Sheard, T. and Fegaras, L. A fold for all seasons. In *Proceedings, Conference on Functional Programming and Computer Architecture*, 1993, pp. 233–242.
23. Takano, A. and Meijer, E. Shortcut deforestation in calculational form. In *Proceedings, Conference on Functional Programming and Computer Architecture*, 1995, pp. 324–333.
24. Tarski, A. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5** (1955) 285–309.
25. Wadler, P. Theorems for free! In *Proceedings, Conference on Functional Programming and Computer Architecture*, 1989, pp. 347–359.