Under consideration for publication in Math. Struct. in Comp. Science

On Proving the Correctness of Program Transformations Based on Free Theorems for Higher-order Polymorphic Calculi

Patricia Johann

Department of Computer Science, Rutgers University, Camden, NJ 08102 USA,
pjohann@crab.rutgers.edu

Received

A number of program transformations currently of interest can be derived from Wadler's "free theorems" for calculi approximating modern functional languages. Although delicate but fundamental issues arise in proving the correctness of free theorems-based program transformations, these issues are usually left unaddressed in correctness proofs appearing in the literature. As a result, most such proofs are incomplete, and most free theorems-based transformations are applied to programs in calculi for which they are not actually known to be correct.
The purpose of this paper is three-fold. First, we raise and clarify some of the issues that must be addressed when constructing correctness proofs for free theorems-based program transformations. Second, we offer a principled approach to developing such proofs. Third, we use Pitts' recent work on parametricity and observational equivalence to show how our approach can be used to give the first proof that transformations based on the Acid Rain theorems preserve observational equivalence of programs in a polymorphic lambda calculus supporting FPC-style fixpoints and algebraic data types. Correctness of the `foldr-build` rule, the `destroy-unfoldr` rule, and the hylofusion program transformation for this calculus follows immediately. The same approach is expected to yield complete correctness proofs for free theorems-based transformations in calculi which even more closely resemble languages with which programmers are concerned in practice.

1. Introduction

How can a program transformation be proved correct? In this paper, we consider this question for program transformations for higher-order polymorphic lambda calculi which approximate modern functional languages. Because many program transformations currently of interest to functional programmers — including `foldr-build` (*i.e.*, short cut) fusion (Gill 1996; Gill *et al.* 1993), `destroy-unfoldr` fusion (Svenningsson 2002), hylofusion (Hu *et al.* 1996; Onoue *et al.* 1997), and their analogues for non-list algebraic data types (Johann 2002b; Takano and Meijer 1995) — are inspired by various "free theorems" (Wadler 1989), we focus particularly on correctness proofs for free theorem-based program transformations in approximating calculi.[†]

The contribution of this paper is three-fold. First, we argue that delicate but fundamental issues arise when constructing correctness proofs for free theorems-based program transformations. Since these issues are typically overlooked in the literature, the justifications for such transformations

[†] As detailed in Section 2.2 below, a free theorem is an equational consequence of parametricity. The term 'free' used in this context should not be confused with the notion of 'free' used in algebra.

which are offered there are almost always incomplete. As a result, most transformations based on free theorems are applied to programs in languages for which they are not actually known to be correct.

Second, we offer a principled approach to developing complete correctness proofs for free theorems-based transformations. At the heart of our approach is the notion of a *parametric model which preserves and reflects observational equivalence* in a calculus.[‡] We argue that a calculus must admit a parametric model in order for free theorems to hold for it, and that it must admit a *model which reflects observational equivalence* — *i.e.*, a model which distinguishes terms which are not observationally equivalent — if program transformations based on those theorems are to be provably correct for it. We show further that if a calculus admits a parametric model which both preserves *and* reflects observational equivalence — *i.e.*, whose induced notion of term equivalence coincides precisely with observational equivalence — then this model can serve as a "universal correctness testbed" for free theorems-based program transformations. Indeed, if a transformation based on free theorems can be proved correct by appealing to *any* model of the calculus, then it can be proved correct by appealing to a parametric model which both preserves and reflects observational equivalence.

Finally, we appeal to the parametric model of PolyFix observational equivalence constructed by Pitts (Pitts 1998; 2000) to show how our approach can be used to give a proof of the Acid Rain theorems (Takano and Meijer 1995) for that calculus. This model, described immediately following Proposition 4.9, both preserves and reflects PolyFix observational equivalence. The Acid Rain theorems are very general statements asserting equivalence between programs which produce and consume data structures in uniform ways, on the one hand, and certain programs which can be derived from them but which avoid the manipulation of intermediate data structures, on the other. PolyFix is a non-strict polymorphic lambda calculus which extends the Girard-Reynolds polymorphic lambda calculus with FPC-style fixpoints and 'lazy' algebraic data types, in which observation of evaluation is permitted only at those types. Our result constitutes the first proof that the Acid Rain theorems preserve observational equivalence of programs in a higher-order polymorphic calculus supporting fixpoints and algebraic data types. Correctness of the `foldr-build` rule, the `destroy-unfoldr` rule, and hylofusion for PolyFix — all of which derive from these free theorems — follows immediately.

The remainder of this paper is organized as follows. In Section 2, we introduce the issues surrounding the correctness of free theorems-based program transformations with which we will be concerned in this paper, and describe a generally applicable approach to addressing them. We further observe that this approach, together with Pitts' parametric model of PolyFix observational equivalence, can be used to prove the correctness of free theorems-based transformations on programs in that calculus. The bulk of this paper is then devoted to making this observation precise. Toward that end, Section 3 describes the syntax and operational semantics of PolyFix, and introduces an appropriate notion of observational equivalence of PolyFix terms. Section 4 introduces some auxiliary notions which are used to construct the parametric model of PolyFix observational equivalence discussed in this paper. This model is obtained simply by identifying observationally equivalent PolyFix terms; proof that it is parametric is given in (Pitts 1998; 2000) and sketched below. In Section 5 we derive PolyFix analogues of some free theorems from (Wadler 1989). Importantly, the functions to which

[‡] In the semantics literature — see, *e.g.*, (Mitchell 1996) or (Reynolds 1998) — a model is said to be *computationally adequate* (or *sound*) if it reflects observational equivalence, and *fully abstract* if it both reflects and preserves observational equivalence. We use the above terminology because it highlights the connection between equivalence in the model and observational equivalence, and because it allows us to discuss preservation and reflection of observational equivalence independently of one another.

these analogues pertain manipulate true algebraic data structures, rather than the functional representations of these structures manipulated by their counterparts in Wadler's paper. In Section 6 we show how Pitts' machinery for investigating parametric polymorphism and observational equivalence can be used to prove correct the Acid Rain theorems for PolyFix. Although we prove the theorems only for list-manipulating PolyFix functions, the fact that PolyFix supports arbitrary 'lazy' algebraic data structures ensures that our results are generalizable in a straightforward manner to non-list algebraic data types as well. Section 7 concludes and offers some suggestions for future investigation.

2. The issues

2.1. *Correctness and observational equivalence*

In this paper we are interested in the correctness of program transformations based on free theorems. It is therefore appropriate to begin by asking a general question about program correctness: What does it mean to say that a program transformation is correct? More fundamentally, what does it even mean to say that we have a program transformation $L = R$ in the first place?

A given calculus can, of course, support many different notions of program equivalence. Moreover, different program transformations may preserve some notions of equivalence but not others. For these reasons, it is clearest to use notation for program transformations which explicitly specifies the notion of program equivalence to be preserved. Implicit in the notation $L = R$ for a transformation on programs in a given calculus is the assertion that an expression matching L in any program in that calculus can be replaced by the corresponding instance of R *without changing the observable behavior of the program.* In this paper, we will write $L =_{obs} R$ for a program transformation which preserves observational equivalence. This notation reflects our expectation that any reasonable notion $=_{obs}$ of observational equivalence of terms will necessarily be a congruence.

To prove the correctness of a given program transformation for a particular calculus of interest, it is necessary to demonstrate that contextual replacement of expressions according to the transformation preserves the observational equivalence of its programs. This can be accomplished by exhibiting a model reflecting observational equivalence for the calculus, and then showing that the left- and right-hand sides of the transformation under consideration have the same interpretation in the model. While a model reflecting observational equivalence in a calculus may indeed identify precisely those terms which are observationally equivalent — *i.e.*, may both preserve *and* reflect observational equivalence — this is not required: the notion of equivalence induced by a model reflecting observational equivalence need only distinguish terms having different observable behavior and be closed under congruence. These requirements ensure that each instance of the left-hand side of the transformation is observationally equivalent to the corresponding instance of its right-hand side, and that replacement, in any context, of any term by an observationally equivalent one preserves observational equivalence.

It is possible to argue the correctness of different program transformations on the basis of different models reflecting observational equivalence. These models might even be produced on a case-by-case basis. But since a model preserving and reflecting observational equivalence for a calculus identifies the left- and right-hand sides of a transformation whenever any model reflecting it does, we see that it is no harder to demonstrate the correctness of a program transformation via a model which both preserves *and* reflects observational equivalence when one exists than it is to do so via a model which simply reflects observational equivalence. This observation eliminates the need for *ad hoc* construction of models reflecting observational equivalence, allowing us to prove correctness of program transformations by appealing instead to a "universal" such model for each calculus.

2.2. *Free theorems*

Our focus in this paper is not on the correctness of just any program transformations, but rather on the correctness of those transformations which have their basis in free theorems for various calculi. We must therefore understand the circumstances under which such transformations may be at our disposal, and so we ask: What is a free theorem? When do free theorems hold for a given calculus?

A free theorem (Wadler 1989) is an equivalence between two terms of the same type in a polymorphic calculus.[§] Free theorems record constraints on the behavior of polymorphic functions in a calculus, and derive from the observation that a polymorphic function must always use the same algorithm to compute its result, regardless of the type at which it is applied. The theorems are "free" in the sense that they are immediate consequences of the syntactic structure of the types of the polymorphic functions whose behavior they describe, and that they can often be read off directly from that structure.

Free theorems hold only for calculi which admit parametric models. A model for a calculus is said to be *parametric* if the notion of equivalence it induces (by identifying terms which have the same interpretation in the model) is the same as the notion of equivalence induced by some Reynolds-style logical relation (Reynolds 1983). A *logical relation* is a type-stratified relation on terms which is constructed in a syntax-directed, bottom-up way via a kind of "induction" on the structure of the types in a calculus. The key to constructing a logical relation is to interpret each base type as a relation between terms of that type, and to specify, for each type constructor, a corresponding relational action which propagates these relations up the type hierarchy. This is done in such a way that polymorphic functions are related if they "map related arguments to related results." Free theorems all derive from the key observation about logical relations, namely that every closed term of closed type is related to itself by the relational interpretation of its type.

Establishing that an equivalence between two closed terms of the same closed type in a polymorphic calculus is a free theorem is achieved by first exhibiting a parametric model for the calculus, then observing that every closed term of the type in question is related to itself by the relational interpretation of that type, and then "unwinding" this observation according to the relational interpretations of types to get a relationship between two terms which can be judiciously instantiated to infer that the two original terms are related in the model. (See, for example, (Wadler 1988), (Gill *et al.* 1995), and (Takano and Meijer 1995).) Non-trivial side conditions sometimes arise during this process, and these must also be shown to hold in order for the theorems to be applicable.

A given calculus might admit a number of parametric models, each giving rise to a different collection of free theorems. Different equivalences can be shown to be free theorems by appealing to different parametric models, and these models can be constructed on a case-by-case basis. Wadler, for example, derives his famous free theorems for the Girard-Reynolds polymorphic lambda calculus $\lambda^{\forall}$ (Wadler 1988) by appealing to a parametric model for it which is based on the frame semantics of (Bruce and Meyer 1984) and (Mitchell and Meyer 1985). Another parametric model for $\lambda^{\forall}$ is given in (Breazu-Tannen and Coquand 1988).

Combining the (orthogonal) observations that each *parametric model* for a polymorphic calculus of interest gives rise to free theorems for it, and that a *model which preserves and reflects observational equivalence* is "universal" for proving the correctness of program transformations for such a calculus, we see that when the correctness of free theorems-based program transformations for a polymorphic

[§] There are, in fact, free theorems which are not equivalences, such as the semantic approximations studied in (Reynolds 1983) and in (Johann and Voigtländer 2004). But the free theorems most commonly considered are equivalences, as are all of the free theorems in (Wadler 1989) and all of those with which we will be concerned in this paper.

calculus can be argued at all, then it can be argued by appealing to a parametric model which preserves and respects observational equivalence. For any calculus which supports one, such a model is "universal" for the correctness of free theorems-based program transformations.

2.3. *Free theorems-based transformations for PolyFix*

The free theorems-based transformations whose correctness we study in this paper transform programs not in $\lambda^{\forall}$ itself, but rather in calculi which more closely approximate modern functional languages. We are particularly interested in the correctness for such calculi of transformations which eliminate intermediate data structures from modularly constructed programs to produce more efficient monolithic equivalents. Examples of such transformations are the `foldr-build` rule, the `destroy-unfoldr` rule, and the hylofusion transformation.

Most justifications for free theorems-based program transformations for approximating calculi which appear in the literature appeal to Wadler's free theorems. Such appeals are, however, problematic: approximating calculi are typically obtained by adding features — fixpoint combinators and algebraic data types are common — to $\lambda^{\forall}$, whereas Wadler's theorems apply *only* to $\lambda^{\forall}$ itself. Moreover, analogues of Wadler's free theorems are not known, *a priori*, to hold for such extensions of $\lambda^{\forall}$. It is, of course, entirely possible that such analogues do not hold for them at all.

Even when free theorems do hold for a particular extension of $\lambda^{\forall}$, there is no reason to suspect that a parametric model in which they hold reflects observational equivalence. This property of a parametric model must be shown explicitly in order to deduce the correctness of any program transformation based on free theorems. In particular, although there is a substantial body of literature on parametricity for $\lambda^{\forall}$ itself — see, *e.g.*, (Abadi *et al.* 1993), (Plotkin and Abadi 1993), and (Reynolds and Plotkin 1993) — these works do not tie parametricity to the observational behavior of terms in $\lambda^{\forall}$. Consequently, the results they report are not suitable for deducing the correctness of free theorems-based program transformations even for $\lambda^{\forall}$, let alone for being extended for that purpose for PolyFix.

In recent work on parametric polymorphism and observational equivalence (Pitts 1998; 2000), Pitts uses an operationally-based logical relations technique to construct a parametric model which both preserves and reflects observational equivalence for PolyFix, a non-strict polymorphic lambda calculus which extends $\lambda^{\forall}$ with FPC-style fixpoints and 'lazy' algebraic data types, and in which observation of evaluation is permitted only at those types.[¶] More specifically, Pitts introduces the notion of a frame stack to describe the contexts in which PolyFix terms can be evaluated, uses frame stacks to give a purely structural characterization of PolyFix termination, and then uses this characterization of termination to define a Reynolds-style logical relation which induces PolyFix observational equivalence. Pitts not only ties the observational behavior of PolyFix programs into the model's underlying relation, but also provides a good deal of useful technical machinery for investigating observational equivalence.

The remainder of this paper is devoted to showing how Pitts' parametric model of PolyFix observational equivalence can be used to establish analogues of Wadler's free theorems for that calculus,

[¶] In (Pitts 2000) this construction and its consequences are worked out in detail for PolyPCF, a version of PolyFix which supports no algebraic data types other than lists. Since PolyFix is obtained from PolyPCF only by adding in non-list algebraic data types, and since no special properties of the list data type are used in (Pitts 2000), it is completely straightforward — if notationally intensive — to develop analogous results for PolyFix to those from (Pitts 2000) for PolyPCF. Indeed, (Pitts 1998) is an incomplete version of precisely such a development. We are therefore justified in appealing to PolyFix analogues of results from (Pitts 2000) throughout this paper.

as well as to prove the correctness of program transformations which derive from them. In particular, we use Pitts' characterization of PolyFix observational equivalence (given in Section 4 below) to prove correct, for PolyFix, the Acid Rain Theorem for Catamorphisms and the Acid Rain Theorem for Anamorphisms. Correctness of the `foldr-build` rule, the `destroy-unfoldr` rule, hylofusion, and their analogues for non-list algebraic data types follows.

Since the `destroy-unfoldr` rule is just an alternate presentation of the Acid Rain Theorem for Anamorphisms, our proof carries out — at least for PolyFix — the "substantial exercise" (Svenningsson 2002) of showing that Pitts' model can be used to prove its correctness. That this model might be so used is suggested by its previous successful use in proving the correctness for PolyFix of short cut fusion and some of its generalizations which are also proper instances of the Acid Rain Theorem for Catamorphisms (Johann 2002a; 2002b). To our knowledge, no complete proof that either the Acid Rain Theorem for Catamorphisms or the Acid Rain Theorem for Anamorphisms preserves observational equivalence of programs has previously appeared in the literature.[‖]

Although we do not offer a formal proof of this fact, it is not difficult to see that the approach developed here can be used to prove the correctness, for any calculus admitting construction of a parametric model preserving and reflecting observational equivalence, of any free theorems-based transformation on programs in that calculus. We thus have at our disposal a promising approach to proving the correctness of such transformations for production-quality functional languages. While deeply theoretical, Pitts' work is thus of enormous practical interest as well.

Like Wadler's free theorems, the theorems which we consider here do not apply directly to the full-scale languages to which they must ultimately be extended if they are to be truly useful. Nevertheless, this paper does mark progress on bridging the gap between the theories of parametricity and observational equivalence on the one hand, and the use in practice of program transformations and other free theorems guaranteed by these theories, on the other.

3. PolyFix

In this section we introduce PolyFix, the polymorphic lambda calculus for which we formalize, and prove the correctness of, the Acid Rain theorems for lists. We also outline those aspects of observational equivalence for PolyFix terms which are needed in this endeavor. PolyFix was introduced in (Pitts 1998), and those aspects of the calculus relevant to proving the correctness of program transformations are summarized below.

3.1. *Syntax*

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 (Fiore and Plotkin 1994; Reynolds 1974). 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, PolyFix is assumed to support only the latter.

[‖] Takano and Meijer offer proofs of both Acid Rain theorems for a calculus with higher-order functions, fixpoints, and algebraic data types, but, unfortunately, these proofs appeal to Wadler's free theorems for $\lambda^{\forall}$, rather than to analogues of those theorems for the calculus in question. Moreover, the question of whether or not free theorems describe the behavior of programs in Takano and Meijer's calculus up to observational equivalence is neither raised nor resolved.

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

Fig. 1. Syntax of PolyFix

The syntax of PolyFix types and terms is given in Figure 1. The syntax

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

is anonymous notation for a recursive data type δ satisfying the fixed point equation

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

Injections into this sum are named explicitly by δ's constructors $\texttt{c}_1$,...,$\texttt{c}_m$; we write $\texttt{c}_i^\delta$ to emphasize that the constructor $\texttt{c}_i$ is associated with the data type δ. Terms of type δ are introduced using δ's constructors and eliminated using case expressions. The types τ_{ij} 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 given in Definition 3.2 below.

Example 3.1. The following are PolyFix data types:

$$\texttt{data}(\alpha = \texttt{Pr}\ \tau\ \tau')$$
$$\texttt{data}(\alpha = \texttt{Just}\ \tau \mid \texttt{Nothing})$$
$$\texttt{data}(\alpha = \texttt{Cons}\ \tau\ \alpha \mid \texttt{Nil})$$

We denote these types by *Pair* $\tau\ \tau'$, *Maybe* τ, and *List* τ, respectively.

As the definitions of *Pair* $\tau\ \tau'$ and *Maybe* τ illustrate, recursive data types can be recursive in the trivial sense. In addition to being anonymous, PolyFix data types can 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.

PolyFix 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 $M\overline{s_n}$ for the n-fold application $Ms_1...s_n$. We similarly write $\lambda\overline{x_n} : \overline{\tau_n}.\, M$ for the n-fold abstraction $\lambda x_1 : \tau_1.\ ...\lambda x_n : \tau_n.\, M$.

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

To be well-formed we require a data type as in (1) to have distinct data constructors $\mathtt{c}_i$, and to be algebraic in the sense of the next definition. The data types *Pair* τ τ', *Maybe* τ, and *List* τ are all algebraic.

Definition 3.2. 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 given by

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

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 $j = 1, ..., k_i$.

We restrict our attention to PolyFix terms which are typeable. The type assignment relation for PolyFix is standard; it is given in Figure 2. A *typing environment* Γ is a pair A, D with A a finite set of type variables, and D a function defined on a finite set $dom(D)$ of variables which maps each $x \in dom(D)$ 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 $\Gamma = A, D$ by extending the function D to map $x \notin dom(D)$ to τ, and Γ, α for the type environment obtained by extending A with a type variable $\alpha \notin A$. Implicit in the notation $\Gamma \vdash M : \tau$ are the assumptions that $\Gamma = A, D$, that $ftv(M) \subseteq A$, that $ftv(\tau) \subseteq A$, and that $fv(M) \subseteq dom(D)$. Note that if $\Gamma = A, D$ and $\Gamma, \alpha \vdash M : \tau$ for some M and τ, then the fact that $\alpha \notin A$ ensures that α does not appear among the free type variables of Γ. The implicit assumptions thus render unnecessary the requirement that α not be among the free variables of Γ that usually accompanies the rule in Figure 2 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*. If $\tau \in$ *Typ*, then the set of closed PolyFix terms M for which $\emptyset, \emptyset \vdash M : \tau$ is denoted $Term(\tau)$.

The (closed) PolyFix terms in Figure 3 appear in the Acid Rain theorems in Section 6. We write $\mathtt{Nil}_\tau$ and $\mathtt{Cons}_\tau$ for $\mathtt{Nil}^{List\,\tau}$ and $\mathtt{Cons}^{List\,\tau}$, respectively, and similarly for the constructors of the *Pair* and *Maybe* types. We write l_τ for the Church encoding $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$ of the data type *List* τ.

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

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

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

$$\frac{\Gamma \vdash M_j:\tau_j[\delta/\alpha] \qquad j=1,..,k_i}{\Gamma \vdash \mathtt{c}_i^{\delta} M_1 ... M_{k_i}:\delta} \text{ if } \delta \text{ is } \mathtt{data}(\alpha = \mathtt{c}_1\overline{\tau_{1k_1}} \mid ... \mid \mathtt{c}_m\overline{\tau_{mk_m}})$$

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

Fig. 2. PolyFix type assignment

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

$$\begin{array}{lcl}
\mathtt{build+} & : & \forall\alpha.\forall\gamma.(\forall\beta.\beta\to(\alpha\to\beta\to\beta)\to\gamma\to\beta)\to\gamma\to \mathit{List}\,\alpha \\
\mathtt{build+} & = & \Lambda\alpha.\Lambda\gamma.\,\lambda g:\forall\beta.\,\beta\to(\alpha\to\beta\to\beta)\to\gamma\to\beta.\,\lambda e:\gamma. \\
 & & \quad g\ (\mathit{List}\,\alpha)\ \mathtt{Nil}_\alpha\ (\lambda h:\alpha.\,\lambda t:\mathit{List}\,\alpha.\,\mathtt{Cons}_\alpha\ h\ t)\ e
\end{array}$$

$$\begin{array}{lcl}
\mathtt{unfoldr} & : & \forall\alpha.\forall\beta.(\beta\to \mathit{Maybe}\,(\mathit{Pair}\ \alpha\ \beta))\to\beta\to \mathit{List}\,\alpha \\
\mathtt{unfoldr} & = & \Lambda\alpha.\Lambda\beta.\mathit{fix}(\lambda h:(\beta\to \mathit{Maybe}\,(\mathit{Pair}\ \alpha\ \beta))\to\beta\to \mathit{List}\ \alpha. \\
 & & \qquad \lambda p:\beta\to \mathit{Maybe}\,(\mathit{Pair}\ \alpha\ \beta).\,\lambda e:\beta. \\
 & & \qquad\quad \mathtt{case}\ p\ e\ \mathtt{of}\ \{\mathtt{Nothing}_{\mathit{Pair}\ \alpha\ \beta} \Rightarrow \mathtt{Nil}_\alpha \mid \\
 & & \qquad\qquad\qquad \mathtt{Just}_{\mathit{Pair}\ \alpha\ \beta}(\mathtt{Pr}_{\alpha\ \beta}\ x\ y) \Rightarrow \mathtt{Cons}_\alpha\ x\ (h\ \alpha\ \beta\ p\ y)\})
\end{array}$$

$$\begin{array}{lcl}
\mathtt{destroy} & : & \forall\beta.\forall\gamma.(\forall\alpha.\,(\alpha\to \mathit{Maybe}\,(\mathit{Pair}\ \beta\ \alpha))\to\alpha\to\gamma)\to \mathit{List}\,\beta\to\gamma \\
\mathtt{destroy} & = & \Lambda\beta.\Lambda\gamma.\lambda g:\forall\alpha.\,(\alpha\to \mathit{Maybe}\,(\mathit{Pair}\ \beta\ \alpha))\to\alpha\to\gamma.\,\lambda xs:\mathit{List}\,\beta. \\
 & & \quad g\ (\mathit{List}\,\beta)\ (\mathtt{listpsi}\ \beta)\ xs
\end{array}$$

$$\begin{array}{lcl}
\mathtt{listpsi} & : & \forall\alpha.\mathit{List}\ \alpha\to \mathit{Maybe}\,(\mathit{Pair}\ \alpha\ (\mathit{List}\ \alpha)) \\
\mathtt{listpsi} & = & \Lambda\alpha.\lambda xs:\mathit{List}\,\alpha.\ \mathtt{case}\ xs\ \mathtt{of}\ \{\mathtt{Nil}_\alpha \Rightarrow \mathtt{Nothing}_{\mathit{Pair}\ \alpha\ (\mathit{List}\,\alpha)} \mid \\
 & & \qquad\qquad \mathtt{Cons}_\alpha\ z\ zs \Rightarrow \mathtt{Just}_{\mathit{Pair}\ \alpha\ (\mathit{List}\,\alpha)}(\mathtt{Pr}_{\alpha\ (\mathit{List}\,\alpha)}\ z\ zs)\}
\end{array}$$

Fig. 3. PolyFix terms

3.2. *Operational semantics*

The operational semantics of PolyFix is given by the evaluation relation in Figure 4. It relates closed terms M to values V of the same closed types. The set of PolyFix *values* is given by

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

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

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

$$\frac{D \Downarrow \mathtt{c}_i^{\delta}\overline{M_{k_i}} \qquad M[\overline{M_{k_i}}/\overline{x_{k_i}}] \Downarrow V}{\mathtt{case}\ D\ \mathtt{of}\ \{... \mid \mathtt{c}_i\overline{x_{k_i}} \Rightarrow M \mid ...\} \Downarrow V} \text{ if } \delta \text{ is } \mathtt{data}(\alpha = \mathtt{c}_1\overline{\tau_{1k_1}} \mid ... \mid \mathtt{c}_m\overline{\tau_{mk_m}})$$

Fig. 4. PolyFix evaluation relation

We write $M \Downarrow V$ if M evaluates to V, and $M \Downarrow$ to indicate that $M \Downarrow V$ for some value V. According to Figure 4, PolyFix 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. Indeed, if Ω is the "polymorphic bottom" $\Lambda\alpha.\mathtt{fix}(\lambda x : \alpha.x)$, then $\Omega\,\tau$ diverges for every type τ, i.e., for no type τ is there a value V such that $\Omega\,\tau \Downarrow V$. In fact, Ω is the only closed term of type $\forall\alpha.\alpha$ up to observational equivalence. In other words, if M is any other closed term of type $\forall\alpha.\alpha$ then $M =_{obs} \Omega : \forall\alpha.\alpha$, where $=_{obs}$ is as defined in the next subsection.

If M and M' are PolyFix terms, then all terms of the form $\mathtt{Pr}\ M\ M'$, `Nothing`, $\mathtt{Just}\ M$, `Nil`, and $\mathtt{Cons}\ M\ M'$ are PolyFix values.

3.3. *Observational equivalence*

Informally, two terms in a programming language are observationally equivalent if they are interchangeable in any program with no change in observable behavior when the resulting programs are executed. If, as in (Pitts 1998), we 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, then we can formalize this by defining two PolyFix terms M_1 and M_2 such that $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$ to be *observationally 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,

$$\mathcal{M}[M_1] \Downarrow \quad \text{iff} \quad \mathcal{M}[M_2] \Downarrow$$

In other words, two PolyFix terms are observationally equivalent if they exhibit the same ground termination behavior in context. As usual, an *evaluation 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 $\Gamma \vdash M_1 =_{obs} M_2 : \tau$ to indicate that M_1 and M_2 are observationally equivalent terms of type τ with respect to Γ. If M_1 and M_2 are closed terms and τ is a closed type, then we write $M_1 =_{obs} M_2 : \tau$ instead of $\emptyset, \emptyset \vdash M_1 =_{obs} M_2 : \tau$. In this case we say simply that M_1 and M_2 are *observationally equivalent*.

4. Parametricity

Pitts' construction of a parametric model which preserves and reflects PolyFix observational equivalence puts the operationally-based logical relations machinery developed in (Pitts 1998; 2000) to good use. Pitts uses the notion of $\top\top$-closure of term relations to identify those which are admissible for fixpoint induction and, thereby, to define the relational actions which give rise to parametric

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

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

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

Fig. 5. Frame stack type judgements

models for PolyFix. Since $\top\top$-closure is defined in terms of the *structural termination relation* $\top$ on PolyFix terms, and since observational equivalence is defined in terms of the termination properties of PolyFix terms, this allows him to tie the theory of PolyFix observational equivalence into a notion of relational parametricity which is analogous to that introduced by Reynolds for the pure polymorphic lambda calculus (Reynolds 1983). This notion of relational parametricity makes precise the sense in which a PolyFix function is or is not "sufficiently polymorphic" to support observational equivalence-preserving free theorems.

After introducing the structural termination relation $\top$ and the notion of $\top\top$-closure in Section 4.1, they are used to define, in Section 4.2, the relational actions for which the Parametricity Theorem for PolyFix is formalized. In the interest of brevity we present here only those portions of the theory of $\top\top$-closure necessary for proving the correctness of free theorems-based program transformations. For a more detailed discussion of $\top\top$-closure, the reader is referred to (Abadi 2000) and (Pitts 2000).

4.1. *Frame stacks and $\top\top$-closed relations*

The notion of $\top\top$-closure is induced by a Galois connection between term relations and relations between evaluation contexts. Pitts recasts evaluation contexts as frame stacks, which aids in their analysis.

Definition 4.1. 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 all free type variables of all expressions occurring in the judgement. The axioms and rules defining this judgement are given in Figure 5.

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. This property is sufficient for our purposes, since the argument types of frame stacks will always be known at the time of use.

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

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

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

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

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

Fig. 6. PolyFix structural termination relation

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

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

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 a stack as follows:

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

Here, $F[M]$ is the term that results from replacing '$-$' by M in the frame F. Note 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

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

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 4.

PolyFix termination is captured by the structural termination relation $(-)\top(-)$ defined in Figure 6: for all closed types τ, all closed algebraic data types δ, all frame stacks $S \in Stack(\tau, \delta)$, and all $M \in Term(\tau)$,

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

Definition 4.2. 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, and is the key ingredient in the definition of $\top\top$-closure.

Definition 4.3. Given any closed types τ and τ', and any $r \in Rel(\tau, \tau')$, define $r^{\top} \in Rel^{\top}(\tau, \tau')$ by

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

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

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

Definition 4.4. A term relation r is said to be $\top\top$-*closed* if $r = r^{\top\top}$.

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

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

This characterization of $\top\top$-closedness will be used in Section 6.3. It is reminiscent of the notion of continuity for relations. This is not surprising, since Abadi (Abadi 2000) has shown that every $\top\top$-closed relation is admissible, i.e., is strict and continuous. He has also established that the converse implication does not hold.

4.2. *Relational actions and parametricity*

We are now in a position to define the relational actions which give rise to parametric models which preserve and reflect PolyFix observational equivalence. The following constructions on term relations describe the ways in which the various PolyFix constructors act on them.

Definition 4.5. Action of $\to$ on term relations: Given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, define $r_1 \to r_2 \in Rel(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$ by

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

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

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

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

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

and

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

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 define the term relation $\mathtt{c}_i r_{i1}...r_{ik_i} \in Rel(\delta, \delta')$ by

$$\mathtt{c}_i r_{i1}...r_{ik_i} \ = \ \{(\mathtt{c}_i^{\delta}\overline{M_{k_i}}, \mathtt{c}_i^{\delta}\overline{M'_{k_i}}) \mid \ \text{for all } j = 1, ..., k_i.\ (M_j, M'_j) \in r_{ij}\}.$$

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

Definition 4.6. A *relational action* Δ for PolyFix comprises a family of mappings

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

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

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

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 and σ' to the term relation $\Delta_\tau(r/\alpha, \overline{r_n}/\overline{\alpha_n}) \in Rel(\tau[\overline{\tau_n}/\overline{\alpha_n}][\sigma/\alpha], \tau[\overline{\tau_n'}/\overline{\alpha_n}][\sigma'/\alpha])$. According to Definition 4.5, we therefore have $\forall r.\, \Delta_\tau(r/\alpha, \overline{r_n}/\overline{\alpha_n}) \in Rel(\forall\alpha.\tau[\overline{\tau_n}/\overline{\alpha_n}], \forall\alpha.\tau[\overline{\tau_n'}/\overline{\alpha_n}])$, as required by Definition 4.6.

Definition 4.7. The relational action ν is given as in Definition 4.6, where the greatest fixed point is taken when defining the relational action at a data type δ in the fourth clause above.

The greatest fixed point of the mapping in the fourth clause of Definition 4.6 exists by Tarski's fixed point theorem (Tarski 1955): each of the sets $Rel(\tau, \tau')$ forms a complete lattice with respect to set inclusion, and the restriction to algebraic data types ensures that the mapping is monotone. The relation ν identifies programs as much as possible, distinguishing them only if there are observable reasons for doing so. This gives a coinductive character to the action of ν at algebraic data types.

Example 4.8. Let τ and τ' be closed types, let $r \in Rel(\tau, \tau')$, and let $r' \in Rel(List\,\tau, List\,\tau')$.

1 The action of `Cons` on term relations is

$$\mathtt{Cons}\, r\, r' \;=\; \{(\mathtt{Cons}_\tau\ H\ T, \mathtt{Cons}_{\tau'}\ H'\ T') \mid (H, H') \in r \text{ and } (T, T') \in r'\}$$

2 The action of `Nil` on term relations is $\mathtt{Nil} = \{(\mathtt{Nil}_\tau, \mathtt{Nil}_{\tau'})\}$
3 Define $1{+}(r \times r') \in Rel(List\,\tau, List\,\tau')$ by

$$1{+}(r \times r') = \{(\mathtt{Cons}_\tau\ H\ T, \mathtt{Cons}_{\tau'}\ H'\ T') \mid (H, H') \in r \text{ and } (T, T') \in r'\} \cup \{(\mathtt{Nil}_\tau, \mathtt{Nil}_{\tau'})\}$$

and write $List\, r$ for the greatest fixed point of the mapping $r' \mapsto (1{+}(r \times r'))^{\top\top}$. Then $\nu_{List\,\tau}(\overline{r_n}/\overline{\alpha_n}) = List\,\nu_\tau(\overline{r_n}/\overline{\alpha_n})$ and, in particular, $\nu_{List\,\alpha}(r/\alpha, \overline{r_n}/\overline{\alpha_n}) = List\,\nu_\alpha(r/\alpha, \overline{r_n}/\overline{\alpha_n}) = List\, r$. Note that, for every relation r, $List\, r$ is $\top\top$-closed and $List\, r^{\top\top} = List\, r$. These observations will be used in Section 5.
4 Since for every pair of appropriately typed stacks S and S', neither $S \top \Omega\,(List\,\tau)$ nor $S' \top \Omega\,(List\,\tau')$ ever holds, and since $(1{+}(r \times r'))^{\top\top}$ is $\top\top$-closed, (2) guarantees that the pair $(\Omega\,(List\,\tau), \Omega\,(List\,\tau'))$ is always in $(1{+}(r \times r'))^{\top\top}$. Thus $(\Omega\,(List\,\tau), \Omega\,(List\,\tau')) \in List\, r$.

Focusing attention on $\nu_\tau()$, we have the following analogue of Corollary 4.1 of (Pitts 2000). This result also appears as Proposition 4.5 in (Pitts 1998).

Proposition 4.9. (Parametricity Theorem for closed PolyFix terms) If ν is a relational action, then for each closed type τ and each closed term M of type τ, $(M, M) \in \nu_\tau()$.

Pitts has actually shown that the notion of program equivalence induced by $\nu_\tau()$ coincides with observational equivalence of closed PolyFix terms at the closed type τ. In fact, he shows a stronger correspondence between ν and observational equivalence: using an appropriate notion of closing substitution to extend ν to a relation $\Gamma \vdash M \,\nu\, M' : \tau$ between open terms, he shows that

$$\Gamma \vdash M =_{obs} M' : \tau \quad \text{iff} \quad \Gamma \vdash M \,\nu\, M' : \tau \tag{3}$$

This result ensures that the identification of observationally equivalent terms will yield a parametric model which preserves and reflects PolyFix observational equivalence. It is this model, consisting of equivalence classes of terms with respect to observational equivalence, with which we will be concerned in the remainder of this paper. In the next section, we will prove the correctness of a number of free theorems-based program transformations by appealing to this model. Correctness of each transformation will be proved by first arguing that, since the model is parametric, it gives rise to a relationship between terms which can be instantiated to establish that the left- and right-hand sides of the free theorem are related by the logical relation underlying the model, and are therefore equivalent in the model. We will then observe that, since the model preserves and reflects observational equivlance, the terms on the left- and right-hand sides of the free theorem must, in fact, be observationally equivalent.

For our purposes the following corollaries of (3) will be particularly useful:

Proposition 4.10. For all closed types τ and closed terms M and M' of type τ,

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

Proposition 4.11. For all terms M and M' of type τ and A of type τ',

$$(\lambda x : \tau'.\, M)A \quad =_{obs} \quad M[A/x] \ : \ \tau \tag{4}$$

$$(\Lambda\alpha.\, M)\tau' \quad =_{obs} \quad M[\tau'/\alpha] \ : \ \tau[\tau'/\alpha] \tag{5}$$

$$\texttt{case}\ \mathtt{c}_i\overline{M_{k_i}}\ \texttt{of}\ \{\ldots \mid \mathtt{c}_i\overline{x_{k_i}} \Rightarrow M' \mid \ldots\} \quad =_{obs} \quad M'[\overline{M_{k_i}}/\overline{x_{k_i}}]) \ : \ \tau \tag{6}$$

$$\texttt{fix}\, M \quad =_{obs} \quad M(\texttt{fix}\, M)) \ : \ \tau \tag{7}$$

5. Some free theorems for PolyFix

The following analogues of examples from (Wadler 1989) illustrate the process of obtaining free theorems for PolyFix. Both take termination into account in their $\top\top$-closedness requirements on the term relations r and s interpreting the quantified type variables α and β. Moreover, both theorems hold for true list-manipulating functions, rather than the corresponding functions considered in (Wadler 1989) which manipulate functional representations of lists.

Example 5.1. Let g be a closed term of type $\forall\alpha.\mathit{List}\,\alpha \to \mathit{List}\,\alpha$. By the Parametricity Theorem, we have

$$(g, g) \in \nu_{\forall\alpha.\mathit{List}\,\alpha\to\mathit{List}\,\alpha}()$$

Applying the definition of ν for $\forall$-types shows that this holds iff for all closed types σ and σ' and for all $r \in \mathit{Rel}(\sigma, \sigma')$,

$$(g\,\sigma, g\,\sigma') \in \nu_{\mathit{List}\,\alpha\to\mathit{List}\,\alpha}(r^{\top\top}/\alpha)$$

The definition of ν for arrow types thus guarantees that for all $(xs, xs') \in \nu_{List\,\alpha}(r^{\top\top}/\alpha)$,

$$(g\,\sigma\,xs, g\,\sigma' xs') \in \nu_{List\,\alpha}(r^{\top\top}/\alpha)$$

i.e., for all $(xs, xs') \in List\, r^{\top\top}$,

$$(g\,\sigma\,xs, g\,\sigma' xs') \in List\, r^{\top\top}$$

Because $List\, r^{\top\top} = List\, r$ for all r, this is the same as the requirement that, for all $(xs, xs') \in List\, r$,

$$(g\,\sigma\,xs, g\,\sigma' xs') \in List\, r$$

Writing $\circ$ for relational composition, we can reformulate this as

$$g\,\sigma' \circ List\, r \subseteq List\, r \circ g\,\sigma$$

If we restrict attention to relations which are functions and write $map\, f$ to denote the PolyFix function which produces a new list by applying the function f to every element of an input list, then we obtain a more familiar formulation of this result: for all closed types σ and σ', and for every function $f : \sigma \to \sigma'$,

$$g\,\sigma' \circ map\, f =_{obs} map\, f \circ g\,\sigma$$

This result can be read as asserting the naturality of g. In (Plotkin and Abadi 1993) it is shown that parametricity implies dinaturality — and, therefore, naturality for first-order functions — in $\lambda^{\forall}$. No analogous result has been established for PolyFix.

Example 5.2. Let g be a closed term of type $\forall\alpha.\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to List\,\alpha \to \beta$. By the Parametricity Theorem, we have

$$(g, g) \in \nu_{\forall\alpha.\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to List\,\alpha\to\beta}()$$

Applying the definition of ν for $\forall$-types twice shows that this holds iff for all closed types σ, σ', τ, and τ', and for all $r \in Rel(\sigma, \sigma')$ and $s \in Rel(\tau, \tau')$,

$$(g\,\sigma\,\tau, g\,\sigma'\tau') \in (r^{\top\top} \to s^{\top\top} \to s^{\top\top}) \to s^{\top\top} \to List\, r^{\top\top} \to s^{\top\top}$$

Applying the definition of $\to$ on relations twice guarantees that, for all $(\oplus, \oplus') \in r^{\top\top} \to s^{\top\top} \to s^{\top\top}$ and $(u, u') \in s^{\top\top}$,

$$(g\,\sigma\,\tau\,(\oplus)\,u,\ g\,\sigma'\tau'(\oplus')\,u') \in List\, r^{\top\top} \to s^{\top\top}$$

Assuming r and s are $\top\top$-closed, and expanding the condition on $(\oplus, \oplus')$, this is the same as requiring that

$$\begin{array}{l}\text{if for all } (x, x') \in r \text{ and } (y, y') \in s,\ (\oplus\, x\, y, \oplus'\, x'\, y') \in s,\\ \quad \text{and if } (u, u') \in s,\\ \quad\quad \text{then } g\,\sigma'\tau'(\oplus')\,u' \circ List\, r \subseteq s \circ g\,\sigma\,\tau\,(\oplus)\,u\end{array}$$

Further restricting attention to relations which are functions yields the following equivalent formulation: for all closed types σ, σ', τ, and τ', and for all $\top\top$-closed $f : \sigma \to \sigma'$ and $h : \tau \to \tau'$,

$$\begin{array}{l}\text{if for all } x : \sigma \text{ and } y : \tau,\ h(x \oplus y) =_{obs} (fx) \oplus' (hy),\\ \quad \text{and if } h\,u = u'\\ \quad\quad \text{then } g\,\sigma'\tau'(\oplus')\,u' \circ map\, f =_{obs} h \circ g\ \sigma\ \tau\ (\oplus)\ u\end{array}$$

6. Free theorems for PolyFix program fusion

In this section we both state precisely and prove the correctness of the Acid Rain theorems for PolyFix. Correctness of the `foldr-build` rule, the `destroy-unfoldr` rule, and the hylofusion trans-

formation for compositions of list-processing PolyFix functions follows immediately. Although we prove the Acid Rain theorems only for list-manipulating functions in this paper, our approach is generalizable to non-list algebraic data types in a straightforward manner, along the same lines as in (Pitts 1998) and (Johann 2002b).

6.1. *The Acid Rain theorems*

Since the PolyFix analogue of Theorem 5.1 of (Pitts 2000) guarantees that observational equivalence for open terms is reducible to observational equivalence for closed terms of closed type, we need state and prove the Acid Rain theorems only for closed terms of closed type.

The Acid Rain Theorem for Catamorphisms generalizes the result type of the function g in the standard `foldr-build` rule. It is given in terms of the standard catamorphism `foldr` for lists, and the generalization `build+` of the standard list-producing function `build`. Operationally, `foldr` takes as input types τ and τ', a replacement term $n : \tau'$ for $\texttt{Nil}_\tau$, a replacement term $c : \tau \to \tau' \to \tau'$ for $\texttt{Cons}_\tau$, and a term xs of type $\mathit{List}\,\tau$. It replaces all (fully applied) occurrences of $\texttt{Cons}_\tau$ in xs by c, and the single (fully applied) occurrence of $\texttt{Nil}_\tau$ in xs by n. The result is a value of type τ'. The function `build+`, on the other hand, takes as input types τ and τ'', a term g of type $\forall\beta.\,\beta \to (\tau \to \beta \to \beta) \to \tau'' \to \beta$, and a term e of type τ''. It returns the term

$$g\,(\mathit{List}\,\tau)\,\texttt{Nil}_\tau\,(\lambda h : \tau.\,\lambda t : \mathit{List}\,\tau.\,\texttt{Cons}_\tau\,h\,t)\,e$$

of type $\mathit{List}\,\tau$. PolyFix definitions of `foldr` and `build+` appear in Figure 3.

The Acid Rain Theorem for Catamorphisms ensures that if g has type $\forall\beta.\,\beta \to (\tau \to \beta \to \beta) \to \tau'' \to \beta$, and if e has type τ'', then any occurrence of $\texttt{foldr}\ \tau\ \tau'\ n\ c\ (\texttt{build+}\ \tau\ \tau''\ g\ e)$ in a program can be replaced by $g\ \tau'\ n\ c\ e$ without changing the observational behavior of the program. It is formalized as

Theorem 6.1. (Acid Rain Theorem for Catamorphisms) Let τ, τ', and τ'' be closed types, and let

$$g : \forall\beta.\,\beta \to (\tau \to \beta \to \beta) \to \tau'' \to \beta,$$

$$n : \tau',$$

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

and

$$e : \tau''$$

be closed terms. Then

$$\texttt{foldr}\ \tau\ \tau'\ n\ c\ (\texttt{build+}\ \tau\ \tau''\ g\ e) =_{obs} g\ \tau'\ n\ c\ e\ :\ \tau'$$

Another way to prove the Acid Rain Theorem for Catamorphisms would be to derive it from the isomorphism Pitts establishes in Example 2.8 of (Pitts 2000) between types of the form $\mathit{List}\,\tau$ and their Church encodings $\forall\alpha.\,\alpha \to (\tau \to \alpha \to \alpha) \to \alpha$. The Acid Rain Theorem for Catamorphisms for non-list algebraic data types can be derived in the same way from corresponding isomorphisms between them and their Church encodings. It may also be possible to derive the Acid Rain Theorem for Anamorphisms, as well as its analogues for non-list data types, directly from similar isomorphisms.

The Acid Rain Theorem for Anamorphisms is the dual of the `foldr-build+` rule. As discussed in (Svenningsson 2002), it can be given in terms of two programming constructs — called `unfoldr`

and `destroy` — which "dualize" `foldr` and `build+`, respectively. The function `destroy` is in turn defined in terms of an auxiliary function `listpsi`.

The function `unfoldr` takes as input types τ and τ'', a term $p : \tau'' \to Maybe\,(Pair\ \tau'\ \tau'')$, and a term $e : \tau''$. It returns the term of type $List\,\tau'$ given by

$$\begin{aligned}\texttt{unfoldr}\,\tau\,\tau''\,p\,e\ =\ \texttt{case}\,p\,e\,\texttt{of}\,\{&\texttt{Nothing}_{Pair\ \tau\ \tau''} \Rightarrow \texttt{Nil}_\tau\ \mid\\ &\texttt{Just}_{Pair\ \tau\ \tau''}(\texttt{Pr}_{\tau\ \tau''}\ x\ y) \Rightarrow \texttt{Cons}_\tau\ x\ (\texttt{unfoldr}\ \tau\ \tau''\ p\ y)\})\end{aligned}$$

The function `destroy`, on the other hand, takes as input types τ and τ', a term g of type $\forall\alpha.\,(\alpha \to Maybe\,(Pair\ \tau\ \alpha)) \to \alpha \to \tau'$, and a term xs of type $List\,\tau$. It returns the element of type τ' given by

$$\texttt{destroy}\ \tau\ \tau'\ g\ = g\ (List\,\tau)\ (\texttt{listpsi}\,\tau)\ xs$$

Here,

$$\begin{aligned}\texttt{listpsi}\ \tau\ \texttt{Nil}_\tau\ &=\ \texttt{Nothing}_{Pair_{\tau\ (List\ \tau)}}\\ \texttt{listpsi}\ \tau\ (\texttt{Cons}_\tau\ z\ zs)\ &=\ \texttt{Just}_{Pair_{\tau\ (List\ \tau)}}(\texttt{Pr}_{\tau\ (List\,\tau)}\ z\ zs)\end{aligned}$$

The definitions of `unfoldr` and `destroy` appear in Figure 3.

The Acid Rain Theorem for Anamorphisms ensures that if $g : \forall\alpha.\,(\alpha \to Maybe\,(Pair\ \tau\ \alpha)) \to \alpha \to \tau'$, if $p : \tau'' \to Maybe\,(Pair\ \tau\ \tau'')$ never returns $\texttt{Just}_{Pair_{\tau\ \tau''}}\,(\Omega\,(Pair\,\tau\,\tau''))$ and admits a "stack equivalent" in the sense indicated in the statement of the theorem, and if $e : \tau''$, then any occurrence of $\texttt{destroy}\ \tau\ \tau'\ g\ (\texttt{unfoldr}\ \tau\ \tau''\ p\ e)$ in a program can be replaced by $g\ \tau''\ p\ e\ :\ \tau'$ without changing the observational behavior of the program. (The requirement that p never returns $\texttt{Just}_{Pair_{\tau\ \tau''}}\,(\Omega\,(Pair\,\tau\,\tau''))$ is an artifact of the particular choice of presentation of `unfoldr` in terms of the type $Maybe\,(Pair\ \tau\ \tau'')$, rather than in terms of a type of the form $\texttt{data}(\alpha\ =\ \texttt{N}\ \mid\ \texttt{J}\ \tau\ \tau'')$. The latter does not contain a "junk" value corresponding to $\texttt{Just}_{Pair_{\tau\ \tau''}}\,(\Omega\,(Pair\,\tau\,\tau''))$.) It is formalized by

Theorem 6.2. (Acid Rain Theorem for Anamorphisms) Let τ, τ', and τ'' be closed types, and let

$$g : \forall\alpha.\,(\alpha \to Maybe\,(Pair\ \tau\ \alpha)) \to \alpha \to \tau'$$

and

$$e : \tau''$$

be closed terms, and let

$$p : \tau'' \to Maybe\,(Pair\ \tau\ \tau'')$$

be a closed term which never returns $\texttt{Just}_{Pair_{\tau\ \tau''}}\,(\Omega\,(Pair\,\tau\,\tau''))$ and for which there exists a stack S_p such that for all $a : \tau''$, $p\,a =_{obs} S_p\,a\ :\ Maybe\,(Pair\ \tau\ \tau'')$. Then

$$\texttt{destroy}\ \tau\ \tau'\ g\ (\texttt{unfoldr}\ \tau\ \tau''\ p\ e) =_{obs} g\ \tau''\ p\ e\ :\ \tau'$$

Note that both conditions on p are necessary. If $g = \forall\alpha.\lambda x.\lambda y.\,\texttt{case}\ x\ y\ \texttt{of}\ \texttt{Just}_{Pair_{\tau\ \alpha}}\ z \to \texttt{Nil}_{\sigma'}$, if $p\ =\ \lambda x.\,\texttt{case}\ x\ \texttt{of}\ \texttt{Nil}_{\sigma''}\ \to\ \texttt{Just}_{Pair_{\tau\ (List\sigma'')}}\,(\Omega\,(Pair\,\tau\,(List\,\sigma'')))$, and if $e\ =\ \texttt{Nil}_{\sigma''}$, then p has a stack equivalent, p does sometimes return $\texttt{Just}_{Pair_{\tau\ (List\sigma'')}}\,(\Omega\,(Pair\,\tau\,(List\,\sigma'')))$, and the terms $\texttt{destroy}\ \tau\ (List\,\sigma')\ g\ (\texttt{unfoldr}\ \tau\ (List\,\sigma'')\ p\ e)$ and $g\ (List\,\sigma'')\ p\ e$ are not observationally equivalent. On the other hand, if $g = \forall\alpha.\lambda x.\lambda y.\,\texttt{case}\ x\,(\Omega\,\alpha)\,\texttt{of}\,\texttt{Nothing}_{Pair_{\tau\ \alpha}} \to \texttt{Nil}_{\sigma'}$, if $p = \lambda x.\,\texttt{Nothing}_{Pair\,\tau\,(List\ \sigma'')}$, and if $e = \texttt{Nil}_{\sigma''}$, then p never returns $\texttt{Just}_{Pair\ \tau\ (List\ \sigma'')}\,(\Omega\,(Pair\,\tau\,(List\,\sigma'')))$, p does not have a stack equivalent, and again $\texttt{destroy}\ \tau\ (List\,\sigma')\ g\ (\texttt{unfoldr}\ \tau\ (List\,\sigma'')\ p\ e)$ and $g\ (List\,\sigma'')\ p\ e$ are not observationally equivalent.

If, as we conjecture, every strict function p admits such a "stack equivalent" S_p, then the above is

precisely the usual Acid Rain Theorem for Anamorphisms. Note that although no conditions on p are explicitly mentioned in either (Takano and Meijer 1995) nor (Svenningsson 2002), the counterexamples above show that the usual Acid Rain Theorem for Anamorphisms must certainly include the requirements that p is strict and, when `unfoldr` is defined in terms of the type $Maybe\,(Pair\,\tau\,\tau'')$, never returns $\mathtt{Just}_{Pair_{\tau\,\tau''}}\,(\Omega\,(Pair\,\tau\,\tau''))$.

6.2. *Acid Rain for Catamorphisms is correct*

Let ν be as in Definition 4.7, and let τ, τ', τ'', and g be as in the statement of Theorem 6.1. Since g and its type are closed, Proposition 4.9 ensures that

$$(g,g) \in \nu_{\forall\beta.\,\beta\to(\tau\to\beta\to\beta)\to\tau''\to\beta}() \tag{8}$$

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

$$(g\,\sigma', g\,\sigma) \in \nu_{\beta\to(\tau\to\beta\to\beta)\to\tau''\to\beta}(r^{\top\top}/\beta)$$

Two-fold application of the definition of ν for arrow types ensures that for all $(n',n) \in r^{\top\top}$, $(c',c) \in \nu_{\tau\to\beta\to\beta}(r^{\top\top}/\beta)$, and $(e',e) \in \nu_\beta()$, (8) holds iff

$$(g\;\tau''\;n'\,c'\,e', g\;\tau'\;n\,c\,e) \in\; r^{\top\top}$$

Expanding the condition on (c',c) shows it equivalent to the assertion that if $(a',a) \in \nu_\tau(r^{\top\top}/\beta)$ and $(b',b) \in r^{\top\top}$, then $(c'\;a'\;b', c\;a\;b) \in r^{\top\top}$. Since (8) holds, we conclude that for all closed types σ' and σ and for all $r \in Rel(\sigma',\sigma)$,

$$\begin{array}{l}\text{if } (n',n) \in r^{\top\top},\\ \quad\text{if } (e',e) \in \nu_{\tau''}(),\\ \qquad\text{and if } (a',a) \in \nu_\tau(r^{\top\top}/\beta) \text{ and } (b',b) \in r^{\top\top} \text{ imply } (c'\;a'\;b', c\;a\;b) \in r^{\top\top},\\ \qquad\quad\text{then } (g\;\tau''\;n'\;c'\;e', g\;\tau'\;n\;c\;e) \in r^{\top\top}\end{array} \tag{9}$$

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

Now consider the instantiation

$$\begin{array}{lcl}\tau'' & = & List\,\tau\\ r & = & \{(g,g') \mid \mathtt{foldr}\,\tau\,\tau'\;n\;c\;g\;=_{obs}\;g' : \tau'\}\\ c' & = & \lambda x.\,\lambda y.\,\mathtt{Cons}\;x\;y\\ n' & = & \mathtt{Nil}\\ e' & = & e\end{array}$$

If we can verify that the hypotheses of (9) hold and that r is $\top\top$-closed, then we may conclude that

$$\mathtt{foldr}\,\tau\,\tau'\;n\;c\;(g\;(List\,\tau)\;\mathtt{Nil}\;(\lambda x.\,\lambda y.\mathtt{Cons}\;x\;y)\;e) =_{obs}\;g\,\tau'\;n\,c\,e\;:\;\tau'$$

Then, since $\mathtt{build+}\;\tau\;\tau''\;g\;e =_{obs} g\;(List\,\tau)\;\mathtt{Nil}\;(\lambda x.\lambda y.\mathtt{Cons}\;x\;y)\;e : List\;\tau'$, we will have proved the correctness of the Acid Rain Theorem for Catamorphisms.

To verify that the hypotheses of (9) hold we first observe that the $\top\top$-closedness of r is proved in (Johann 2002a); the proof uses the techniques of the next subsection. Using this fact, we then note that $\mathtt{foldr}\,\tau\,\tau'\;n\;c\;n' =_{obs} \mathtt{foldr}\,\tau\,\tau'\;n\;c\;\mathtt{Nil} =_{obs} n : \tau'$, *i.e.*, that $(n',n) \in r$. Moreover, since τ and τ'' are closed, $\nu_\tau(r^{\top\top}/\beta)$ is precisely $\nu_\tau()$ and $\nu_{\tau''}(r^{\top\top}/\beta)$ is precisely $\nu_{\tau''}()$. Thus, if $(a',a) \in \nu_\tau(r^{\top\top}/\beta)$ and $(e',e) \in \nu_{\tau''}(r^{\top\top}/\beta)$, then by (3) we have $a' =_{obs} a : \tau$ and $e' =_{obs} e : \tau''$. If, in addition, $(b',b) \in r$, then $\mathtt{foldr}\,\tau\,\tau'\;n\;c\;b' =_{obs} b : \tau'$. Since $=_{obs}$ is a congruence, equivalences

(4) through (7) guarantee that

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

i.e., that $(c'\;a'\;b', c\;a\;b) \in r$. Since all hypotheses of (9) are satisfied and r is $\top\top$-closed, we have that

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

as desired.

6.3. *Acid Rain for Anamorphisms is correct*

Let ν be as in Definition 4.7, and let τ, τ', τ'', and g be as in the statement of Proposition 6.2. Since g and its type are closed, Proposition 4.9 ensures that

$$(g, g) \in \nu_{\forall\alpha.\,(\alpha\to \mathit{Maybe}\,(\mathit{Pair}\;\tau\;\alpha))\to\alpha\to\tau'}() \tag{10}$$

Applying the definition of ν for $\forall$-types shows that (10) holds iff for all closed types σ and σ', and for all $r \in \mathit{Rel}(\sigma', \sigma)$,

$$(g\;\sigma', g\;\sigma) \in \nu_{(\alpha\to \mathit{Maybe}\,(\mathit{Pair}\;\tau\;\alpha))\to\alpha\to\tau'}(r^{\top\top}/\alpha)$$

Two-fold application of the definition of ν for arrow types ensures that, for all

$$(p', p) \in \nu_{\alpha\to \mathit{Maybe}\,(\mathit{Pair}\;\tau\;\alpha)}(r^{\top\top}/\alpha)$$

and

$$(e', e) \in r^{\top\top},$$

(10) holds iff

$$(g\;\sigma'\;p'\;e', g\;\sigma\;p\;e) \in \nu_{\tau'}()$$

Expanding the condition on (p', p) shows that it is equivalent to the assertion that if $(a', a) \in r^{\top\top}$, then $(p'\;a', p\;a) \in \nu_{\mathit{Maybe}\,(\mathit{Pair}\;\tau\;\alpha)}(r^{\top\top}/\alpha)$. But $\nu_{\mathit{Maybe}\,(\mathit{Pair}\;\tau\;\alpha)}(r^{\top\top}/\alpha)$ is

$$\begin{array}{ll} & \{(\texttt{Nothing}, \texttt{Nothing})\} \\ \cup & \{(\texttt{Just}\;(\texttt{Pair}\;w'\;z'), \texttt{Just}\;(\texttt{Pair}\;w\;z)) \mid (w', w) \in \nu_\tau() \text{ and } (z', z) \in r^{\top\top}\} \\ \cup & \{(\texttt{Just}\,(\Omega\;(\mathit{Pair}\,\tau\,\sigma')), \texttt{Just}\,(\Omega\;(\mathit{Pair}\,\tau\,\sigma)))\} \\ \cup & \{(\Omega\;(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\sigma')), \Omega\;(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\sigma)))\} \end{array}$$

Since (10) holds, we conclude that for every closed type σ and for all $r \in \mathit{Rel}(\sigma', \sigma)$,

if $(e', e) \in r^{\top\top}$,
and if $(a', a) \in r^{\top\top}$ implies
$(p'\;a', p\;a) \in \{(\texttt{Nothing}, \texttt{Nothing})\}$
$\cup\, \{(\texttt{Just}\;(\texttt{Pair}\;w'\;z'), \texttt{Just}\;(\texttt{Pair}\;w\;z)) \mid (w', w) \in \nu_\tau() \text{ and } (z', z) \in r^{\top\top}\}$
$\cup\, \{(\texttt{Just}\,(\Omega\;(\mathit{Pair}\,\tau\,\sigma')), \texttt{Just}\,(\Omega\;(\mathit{Pair}\,\tau\,\sigma)))\}$
$\cup\, \{(\Omega\;(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\sigma')), \Omega\;(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\sigma)))\}$
then $(g\;\sigma'\;p'\;e', g\;\sigma\;p\;e) \in \nu_{\tau'}()$ (11)

Note that all of the terms appearing in (11) are closed.

Now consider the instantiation

$$\begin{array}{lcl} \sigma & = & \tau'' \\ \sigma' & = & \mathit{List}\ \tau \\ r & = & \{(M, M') \mid M\ =_{obs}\ \mathtt{unfoldr}\ \tau\ \tau''\ p\ M' : \mathit{List}\ \tau\} \\ p' & = & \mathtt{listpsi}\ \tau \\ e' & = & \mathtt{unfoldr}\ \tau\ \tau''\ p\ e \end{array}$$

If we can verify that the hypotheses of (11) hold, then we may conclude that

$$(g\ (\mathit{List}\,\tau)\ (\mathtt{listpsi}\ \tau)\ (\mathtt{unfoldr}\ \tau\ \tau''\ p\ e),\ g\ \tau''\ p\ e) \in \nu_{\tau'}()$$

and so by (3)

$$g\ (\mathit{List}\,\tau)\ (\mathtt{listpsi}\ \tau)\ (\mathtt{unfoldr}\ \tau\ \tau''\ p\ e)\ =_{obs}\ g\ \tau''\ p\ e\ :\ \tau'$$

Then, observing that $\mathtt{destroy}\ \tau\ \tau'\ g\ xs =_{obs} g\ (\mathit{List}\,\tau)\ \mathtt{listpsi}\ xs\ : \tau'$ and instantiating `xs` with $\mathtt{unfoldr}\ \tau\ \tau''\ p\ e$, we will have proved the correctness of the Acid Rain Theorem for Anamorphisms.

To this end, we first prove that r is $\top\top$-closed. To see this, suppose $(M, M') \in r^{\top\top}$. We want to verify that $M\ =_{obs}\ \mathtt{unfoldr}\ \tau\ \tau''\ p\ M' : \mathit{List}\ \tau$. Note that S_p must have the form $\mathit{Id} \circ S'_p$ for some frame S'_p. Let F be the frame

$$\begin{array}{l} \mathtt{case}\ \_\ \mathtt{of} \\ \quad \{\mathtt{Nothing} \Rightarrow \mathtt{Nil}\ \mid \\ \quad\ \mathtt{Just}\ (\mathtt{Pr}\ x\ y) \Rightarrow \mathtt{Cons}\ x\ (\mathtt{unfoldr}\ \tau\ \tau''\ p\ y)\} \end{array}$$

and let $S \in \mathit{Stack}(\tau'', \mathit{List}\ \tau)$ be the "stack equivalent"

$$S = \mathit{Id} \circ F \circ S'_p$$

of the evaluation context $\mathtt{unfoldr}\,\tau\,\tau''\,p$. Then S is such that for all $N : \tau''$,

$$S\,N\ =_{obs}\ \mathtt{unfoldr}\,\tau\,\tau''\,p\,N\ :\ \mathit{List}\ \tau \tag{12}$$

since

$$\begin{array}{lll} \mathtt{unfoldr}\,\tau\,\tau''\,p\,N & =_{obs} & (\lambda f : \tau'' \to \mathit{Maybe}\,(\mathit{Pair}\ \tau\ \tau'').\ \lambda b : \tau''. \\ & & \quad \mathtt{case}\ f\,b\ \mathtt{of}\ \{\mathtt{Nothing} \Rightarrow \mathtt{Nil}\ \mid \\ & & \qquad \mathtt{Just}\ (\mathtt{Pr}\ x\ y) \Rightarrow \mathtt{Cons}\ x\ (\mathtt{unfoldr}\ \tau\ \tau''\ f\ y)\})\ p\,N \\ & =_{obs} & \mathtt{case}\,p\,N\ \mathtt{of} \\ & & \quad \{\mathtt{Nothing} \Rightarrow \mathtt{Nil}\ \mid \\ & & \quad\ \mathtt{Just}\ (\mathtt{Pr}\ x\ y) \Rightarrow \mathtt{Cons}\ x\ (\mathtt{unfoldr}\,\tau\,\tau''\,p\,y)\} \\ & =_{obs} & (\mathit{Id} \circ F \circ S'_p)\,N \\ & =_{obs} & S\,N\ :\ \mathit{List}\,\tau \end{array}$$

The first equivalence is by (7) and the definition of `unfoldr`, the second is by repeated application of (4) and (6), the third is by the definition of frame stack application, and the fourth is by the definition of S.

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

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

and

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

then

$$(S'@S) \top M\ \text{ iff }\ S' \top (S\,M) \tag{13}$$

Moreover, for any $S' \in \mathit{Stack}(\tau)$, the frame stack $S' \mathbin{@} S$ has the property that for all (N, N') with $\mathtt{unfoldr}\,\tau\,\tau''\,p\,N' =_{obs} N : \mathit{List}\,\tau$,

$$\begin{array}{ll} & (S' \mathbin{@} S) \top N' \\ \text{iff} & S' \top S N' \\ \text{iff} & S' \top \mathtt{unfoldr}\,\tau\,\tau''\,p\,N' \\ \text{iff} & S' \top N \end{array}$$

The first equivalence by (13), and the second is by Proposition 4.10 and (12), and the third is by Proposition 4.10 and the hypothesis that $\mathtt{unfoldr}\,\tau\,\tau''\,p\,N' =_{obs} N : \mathit{List}\,\tau$. Together with (2), the fact that $(M, M') \in r^{\top\top}$ therefore implies that

$$(S' \mathbin{@} S) \top M' \quad \text{iff} \quad S' \top M \tag{14}$$

But then

$$S' \top M \quad \text{iff} \quad (S' \mathbin{@} S) \top M' \quad \text{iff} \quad S' \top S\,M' \quad \text{iff} \quad S' \top \mathtt{unfoldr}\,\tau\,\tau''\,p\,M'$$

Here, the first equivalence is by (14), the second is by (13), and the third is by (12). Since S' was arbitrary we have shown that

$$\text{for all } S' \in \mathit{Stack}(\tau').\ S' \top M \quad \text{iff} \quad S' \top \mathtt{unfoldr}\,\tau\,\tau''\,p\,M'$$

By Proposition 4.10, we therefore have that $M =_{obs} \mathtt{unfoldr}\,\tau\,\tau''\,p\,M' : \tau'$, and thus that r is $\top\top$-closed. (Alternatively, r can be viewed as the "graph" $\{(M, M') \mid M =_{obs} SM')\}$ of S, which is $\top\top$-closed by the analogue for PolyFix of Lemma 6.1 of (Pitts 2000).)

We now use the observation that r is $\top\top$-closed to verify the hypotheses of (11). First observe that $(e', e) \in r$ trivially. Next note that if $(a', a) \in r$, *i.e.*, if $a' =_{obs} \mathtt{unfoldr}\,\tau\,\tau''\,p\,a : \mathit{List}\,\tau$, then

$$\begin{array}{lll} p'a' & = & \mathtt{listpsi}\,\tau\,a' \\ & =_{obs} & \mathtt{listpsi}\,\tau\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,a) \end{array}$$

Since $p\,a$ is not $\mathtt{Just}\,(\Omega\,(\mathit{Pair}\,\tau\,\tau''))$, there are three cases to consider:

— If $p\,a = \mathtt{Nothing}$, then

$$\begin{array}{lll} p'a' & =_{obs} & \mathtt{listpsi}\,\tau\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,a) \\ & =_{obs} & \mathtt{listpsi}\,\tau\,\mathtt{Nil} \\ & =_{obs} & \mathtt{Nothing} \end{array}$$

So $(p'a', pa) \in \{(\mathtt{Nothing}, \mathtt{Nothing})\}$ in this case.

— If $p\,a = \mathtt{Just}\,(\mathtt{Pair}\,b\,c)$, then

$$\begin{array}{lll} p'a' & =_{obs} & \mathtt{listpsi}\,\tau\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,a) \\ & =_{obs} & \mathtt{listpsi}\,\tau\,(\mathtt{Cons}\,b\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,c)) \\ & =_{obs} & \mathtt{Just}\,(\mathtt{Pair}\,b\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,c)) \end{array}$$

Since $(b, b) \in \nu_\tau()$ and $(\mathtt{unfoldr}\ \tau\ \tau''\,p\,c,\ c) \in r$ — both trivially — we have that $(p'a', pa) \in \{(\mathtt{Just}\,(\mathtt{Pair}\,w'\,z'), \mathtt{Just}\,(\mathtt{Pair}\,w\,z)) \mid (w', w) \in \nu_\tau()$ and $(z', z) \in r^{\top\top}\}$, as desired.

— If $p\,a = \Omega\,(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\tau''))$ then

$$\begin{array}{lll} p'a' & =_{obs} & \mathtt{listpsi}\,\tau\,(\mathtt{unfoldr}\,\tau\,\tau''\,p\,a) \\ & =_{obs} & \mathtt{listpsi}\,\tau\,(\Omega\,(\mathit{List}\,\tau)) \\ & =_{obs} & \Omega\,(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,(\mathit{List}\,\tau))) \end{array}$$

Thus $(p'a', pa) \in \{(\Omega\,(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,(\mathit{List}\,\tau))), \Omega\,(\mathit{Maybe}\,(\mathit{Pair}\,\tau\,\tau'')))\}$, as desired.

Since the hypotheses of (11) hold, the theorem is proved.

7. Conclusion and future work

We have observed that, in order to correctly state and prove analogues of Wadler's free theorems for a polymorphic calculus, it suffices to exhibit a *parametric* model for that calculus. We have also observed that, in order to prove the correctness of a program transformation for such a calculus — in particular, to capture the intuition that the transformation preserves observational equivalence of programs — it suffices to exhibit a *model which reflects observational equivalence* for that calculus and then to show that the left-and right-hand-sides of the transformation have the same interpretation in the model. Finally, we have argued that, in order to prove the correctness of a program transformation which derives from free theorems, it suffices to provide a *parametric model which preserves and reflects observational equivalence* — *i.e.*, a model whose relational equivalence coincides with observational equivalence — which appropriately interprets the left- and right-hand-sides of the transformation. Unfortunately, the need to tie the operational semantics of a calculus into the logical relation underlying a parametric model for it has been overlooked in most correctness proofs for free theorems-based program transformations appearing in the literature.

Pitts' construction of a parametric model which preserves and reflects PolyFix observational equivalence provides the basis of a promising approach to proving the correctness of free theorems-based program transformations for that calculus. We have used his model to give the first-ever proof of correctness for the Acid Rain theorems for a calculus with higher-order functions, fixpoints, and algebraic data types. In addition, we have argued that this same approach can be used to prove the correctness of any free theorems-based program transformation for any calculus admitting construction of a Pitts-style parametric model which preserves and reflects observational equivalence.

While PolyFix supports some of the important features of modern functional languages, our results still need to be extended to calculi which more closely resemble such languages if they are to be truly relevant program transformation in practice. Such extensions are the goal of future work.

Another goal of future work is to modify the approach to proving program correctness put forth in this paper to accommodate calculi with non-call-by-name operational semantics. We anticipate the investigation of both 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. Although a parametric model preserving and reflecting observational equivalence for a call-by-value version of PolyPCF appears in (Pitts 1998a), the details for a full call-by-value PolyFix and a 'lazy' PolyFix remain to be established.

Acknowledgments I am grateful to Dan Dougherty for many thoughtful comments and probing questions about this work, and to Janis Voigtländer for ongoing enlightening conversations about parametricity and free theorems. I also thank the anonymous referees for their helpful remarks.

References

Abadi, M. $\top\top$-closed relations and admissibility. Mathematical Structures in Computer Science, 10 (2000) 313-320.

Abadi, M. Cardelli, L., and Curien, P.-L. Formal parametric polymorphism. Theoretical Computer Science, 121 (1993), 9-58.

Breazu-Tannen, V. and Coquand, T. Extensional models for polymorphism. Theoretical Computer Science, 59 (1988) 85-114.

Bruce, K. B. and Meyer, A. R. The semantics of second-order polymorphic lambda calculus. In Semantics of Data Types, LNCS 173 (1984) 131-144.

Fiore, M. and Plotkin, G. An axiomatization of computationally adequate domain theoretic models of FPC. Proceedings, 9th Annual Symposium on Logic in Computer Science, (1994) 92-102.

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

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

Hu, Z., Iwasaki, H., and Takeichi, M. Deriving structural hylomorphisms from recursive definitions. Proceedings, International Conference on Functional Programming, (1996) 73-82.

Johann, P. Short cut fusion is correct. Journal of Functional Programming, 13(4) (2003) 797-814.

Johann, P. A generalization of short-cut fusion and its correctness proof. Higher-Order and Symbolic Computation, 15 (2002) 273-300. An earlier version of this paper appears as Johann, P. Short Cut Fusion: Proved and Improved. Proceedings, Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196 (2001) 47-71.

Johann, P. and Voigtländer, J. Free Theorems in the Presence of *seq*. Proceedings, Conference on Principles of Programming Languages, (2004).

Mitchell, J. C. *Foundations for Programming Languages*. MIT Press, 1996.

Mitchell, J. C. and Meyer, A. R. Second-order logical relations. In Logic of Programs, LNCS 193 (1985) 235-236.

Onoue, Y., Hu, Z., Iwasaki, H., and Takeichi, M. A calculational system HYLO. Proceedings, IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, (1997) 76-106.

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

Pitts, A. Existential types: Logical relations and operational equivalence. Proceedings, International Colloquium on Automata, Languages, and Programming, (1998), 309-326.

Pitts, A. Parametric polymorphism and operational equivalence. Mathematical Structures in Computer Science, 10 (2000) 1-39.

Plotkin, G. D. and Abadi, M. A logic for parametric polymorphism. Proceedings, Conference on Typed Lambda Calculus and Applications, LNCS 664 (1993) 361-375.

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

Reynolds, J. C. Toward a theory of type structure. Proceedings, Paris Colloquium on Programming, LNCS 19 (1974) 408-425.

Reynolds, J. C. Types, abstraction, and parametric polymorphism. Information Processing, 83 (1983) 513-523.

Reynolds, J. C. *Theories of Programming Languages*. Cambridge University Press, 1998.

Reynolds, J.C. and Plotkin, G. D. On functors expressible in the polymorphic typed lambda calculus. Information and Computation, 121 (1993) 411-440.

Svenningsson, J. Shortcut fusion for accumulating parameters and zip-like functions. Proceedings, International Conference on Functional Programming, (2002) 124-132.

Tarski, A. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5 (1955) 285-309.

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