Free Theorems in the Presence of *seq*

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

Janis Voigtländer[*]
Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

Abstract

Parametric polymorphism constrains the behavior of pure functional programs in a way that allows the derivation of interesting theorems about them solely from their types, i.e., virtually for free. Unfortunately, the standard parametricity theorem fails for nonstrict languages supporting a polymorphic strict evaluation primitive like Haskell's *seq*. Contrary to the folklore surrounding *seq* and parametricity, we show that not even quantifying only over strict and bottom-reflecting relations in the $\forall$-clause of the underlying logical relation — and thus restricting the choice of functions with which such relations are instantiated to obtain free theorems to strict and total ones — is sufficient to recover from this failure. By addressing the subtle issues that arise when propagating up the type hierarchy restrictions imposed on a logical relation in order to accommodate the strictness primitive, we provide a parametricity theorem for the subset of Haskell corresponding to a Girard-Reynolds-style calculus with fixpoints, algebraic datatypes, and *seq*. A crucial ingredient of our approach is the use of an asymmetric logical relation, which leads to "inequational" versions of free theorems enriched by preconditions guaranteeing their validity in the described setting. Besides the potential to obtain corresponding preconditions for standard equational free theorems by combining some new inequational ones, the latter also have value in their own right, as is exemplified with a careful analysis of *seq*'s impact on familiar program transformations.

Categories and Subject Descriptors: D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features—*polymorphism*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*control primitives, type structure*

General Terms: Languages, Theory

Keywords: Controlling strict evaluation, correctness proofs, denotational semantics, Haskell, logical relations, parametricity, program transformations, short cut fusion, theorems for free

1 Introduction

Ever since they were first popularized by Wadler [23], *free theorems* have been used to derive program equivalences involving parametric polymorphic functions in programming languages based on the Girard-Reynolds lambda calculus [7, 16]. A free theorem is considered **free** because it can be derived solely from the type of a function, with no knowledge at all of the function's actual definition. In essence, it records a constraint arising from the fact that a parametric polymorphic function must behave uniformly, i.e., must use the same algorithm to compute its result, regardless of the concrete type at which it is instantiated.

Free theorems hold unconditionally for polymorphic functions in the Girard-Reynolds calculus. But for calculi that more closely resemble modern functional languages the story is not so simple. It is well known [11, 23] that adding a fixpoint primitive to a calculus weakens its free theorems by imposing admissibility conditions on (some of the) functions appearing in them. How free theorems fare in the presence of other primitives is less well understood.

We will use the following example to illustrate that new primitives can break free theorems in dramatic and unexpected ways. A free theorem given in Figure 1 of [23] states that for any function

$$\mathit{filter} :: \forall\alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

and appropriately typed p, h, and l the following law holds:

$$\boxed{\mathit{filter}\ p\ (\mathit{map}\ h\ l) = \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l) \qquad (1)}$$

Here *map* h applies the function h to every element of a list, and $\circ$ is function composition. (See Figure 1.)

While Haskell is a nonstrict language — so that a function argument is evaluated only when required — it is sometimes desirable to explicitly force evaluation. This can be done using the polymorphic strict evaluation primitive *seq*, which satisfies the following specification:[1]

$$\begin{aligned}&\mathit{seq} :: \forall\alpha\,\beta.\ \alpha \to \beta \to \beta\\&\mathit{seq}\ \bot\ b = \bot\\&\mathit{seq}\ a\ \ b = b\ ,\ \text{if}\ a \neq \bot\end{aligned}$$

Here $\bot$ is the undefined value corresponding to a nonterminating computation or a runtime error (such as might be obtained as the result of a failed pattern match). The operational behavior of *seq* is to evaluate its first argument to weak head normal form before returning its second argument. It is usually introduced to improve

[*] Research supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-4.

POPL'04, January 14–16, 2004, Venice, Italy.
Copyright 2004 ACM 1-58113-729-X/04/0001 ...$5.00
This is the authors' version of the work. The definitive version is available from http://doi.acm.org/10.1145/964001.964010.

[1] Other means of explicitly introducing strictness in Haskell programs — e.g., strict datatypes — are all defined in terms of *seq*.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Bool} \quad\ \ = \mathsf{False} \mid \mathsf{True}\\
&\textbf{data}\ \mathsf{Maybe}\ \alpha = \mathsf{Nothing} \mid \mathsf{Just}\ \alpha\\[4pt]
&\mathit{map} :: \forall\alpha\,\beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]\\
&\mathit{map}\ h = f\ \textbf{where}\ f\ []\qquad\ \ = []\\
&\qquad\qquad\qquad\quad\ \ f\ (x : \mathit{xs}) = h\ x : f\ \mathit{xs}\\[4pt]
&(\circ) :: \forall\alpha\,\beta\,\gamma.\ (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\\
&f_1 \circ f_2 = (\lambda x \to f_1\ (f_2\ x))\\[4pt]
&\mathit{fix} :: \forall\alpha.\ (\alpha \to \alpha) \to \alpha\\
&\mathit{fix}\ g = g\ (\mathit{fix}\ g)\\[4pt]
&\mathit{id} :: \forall\alpha.\ \alpha \to \alpha\\
&\mathit{id}\ x = x\\[4pt]
&(+\!\!+) :: \forall\alpha.\ [\alpha] \to [\alpha] \to [\alpha]\\
&[]\qquad\ \ +\!\!+\, \mathit{ys} = \mathit{ys}\\
&(x : \mathit{xs}) +\!\!+\, \mathit{ys} = x : (\mathit{xs} +\!\!+\, \mathit{ys})
\end{aligned}
$$

Figure 1. Some Haskell definitions.

performance by avoiding unnecessary evaluation delay or by sequentializing algorithms in parallel implementations [21].

The Haskell 98 report [12] — the language definition — warns that the provision of polymorphic *seq* has important semantic consequences. In fact, if we implement a function with *filter*'s type using *seq* and a fixpoint operator *fix* (defined as in Figure 1) by

$$
\begin{aligned}
&\mathit{filter}\ p = p\ \text{`}\mathit{seq}\text{`}\ (\mathit{fix}\ g)\\
&\quad \textbf{where}\ g\ f\ \mathit{ys} = \textbf{case}\ \mathit{ys}\ \textbf{of}\\
&\qquad\qquad\qquad\qquad []\qquad \to []\\
&\qquad\qquad\qquad\qquad x : \mathit{xs} \to x\ \text{`}\mathit{seq}\text{`}\ \textbf{case}\ p\ x\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{True}\ \to (\mathit{xs}\ \text{`}\mathit{seq}\text{`}\ x) : f\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{False} \to f\ \mathit{xs}
\end{aligned}
$$

then the following four instantiations break law (1), each for a different reason to be discussed below:

$$
\begin{array}{llll}
p = \bot & h = \mathit{id} & l = [] & (\sqsubsetneq)\\
p = (\lambda x \to \mathsf{True}) & h = \bot & l = [0] & (\sqsubsetneq)\\
p = \mathit{id} & h = (\lambda x \to \mathsf{True}) & l = [\bot] & (\sqsupsetneq)\\
p = \mathit{id} & h = (\lambda x \to \mathsf{True}) & l = \mathsf{True} : \bot & (\sqsubsetneq)
\end{array}
$$

In each of these cases, one of the two sides of the law is less defined than the other one, so that the equivalence (1) is incorrect. The direction in which it turns into an inequation with respect to the semantic approximation order is remarked at the end of each line.

Launchbury and Paterson [11] introduced a type system that makes explicit which types contain $\bot$ — the so-called *pointed* types — and therefore support the definition of values by recursion. This type system allows fine control over where the aforementioned admissibility conditions are required in free theorems and where they are not. In the above definition of *filter*, for example, we have used explicit fixpoint recursion expressed via the operator *fix* rather than giving a directly recursive definition. This makes it clear that the recursion here takes place at type $[\alpha] \to [\alpha]$, which is pointed without any condition on α since $[\alpha]$ is. The definition also makes apparent that the **case**-expressions — which produce $\bot$ on selectors that are themselves undefined — have the return type $[\alpha]$, which is again pointed without any condition on α. Thus, Launchbury and Paterson's approach to parametricity in the presence of pointed vs. unpointed types can be used to show that law (1) holds without any conditions on p, h, or l, provided all invocations of *seq* are dropped[2] (in which case we get precisely the *filter* function from Haskell's standard prelude). This shows that it is not the use of recursion or pattern matching in our definition of *filter* that is responsible for the breakdown of law (1). The evil really does reside in *seq*.

More precisely, different ways of using *seq* are responsible for the failure of law (1) for the four instantiations given above. In the first case, the use of *seq* to observe termination at function type makes the left-hand side $\bot$ since $p = \bot$, while there is no such impact on the right-hand side because $p \circ h = \bot \circ \mathit{id} = (\lambda x \to \bot) \neq \bot$. In the second case, the use of *seq* to observe termination at the type over which *filter* is polymorphic finds $h\ 0 = \bot\ 0 = \bot$ in the left-hand side (since the inner *map* applies h to every list element beforehand), while in the right-hand side the corresponding application is on the list element $0 \neq \bot$ itself (to which h is applied only afterwards by the outer *map*, returning an undefined list element but not a completely undefined result list). So here the problem lies with h returning $\bot$ for a non-$\bot$ argument, which makes an essential difference for *seq*. Conversely, an h returning a non-$\bot$ value for the argument $\bot$ lets the law (1) fail the other way round for the third instantiation. Finally, in the fourth instantiation, the use of *seq* on an undefined list tail as first argument introduces a $\bot$ of the type over which *filter* is polymorphic. So, even though — as discussed above — the given definition of *filter* does not use recursion or pattern matching in a way that would require α to be pointed, the associated strictness condition on h creeps in through the back door here, carelessly opened by *seq*.

The failure of free theorems in the presence of *seq* has been noted before (see, e.g., Section 6.2 of [12], Section 5.3 of [13], Appendix B of [22], and discussions on the Haskell mailing list [1]). But the extent to which Haskell's parametricity properties are actually weakened has not been studied thoroughly. Conventional wisdom (expressed, e.g., in [13]) has it that a free theorem remains valid in the presence of *seq* if all of the functions which are chosen to instantiate relations in the statements derived from types (where one is free to make such a choice) are strict and total. For our example this means that law (1) should hold for every strict and total h — a claim which is shown to be incorrect by our first counterexample above. The need for valid criteria for determining when free theorems hold in the presence of *seq* is quite dire: *seq* will not go away by ignoring it, and neither will the demand for rigorous semantic arguments about Haskell programs that rely on parametricity. The major contribution of this paper is to provide precisely such criteria.

Note that our aim here is not to provide a parametric model for full Haskell. This would be out of reach because there is not even a formal semantics for Haskell against which to validate a model. Rather, our aim is to investigate, under the reasonable and widely held assumption that Haskell without *seq* is parametric, exactly how much of the power of free theorems can be retained when *seq* is thrown in. The assumption is justified in part by the recent construction of a parametric model for a nonstrict polymorphic lambda calculus supporting fixpoints and "lazy" algebraic datatypes [14].

The fundamental idea underlying the parametricity properties from which free theorems are derived is to interpret types as relations (as opposed to sets). It is standard to interpret the base types in a language as identity relations and to obtain the interpretations for non-base types by propagating relations up the type hierarchy in a straightforward "extensional" manner. This builds a (type-indexed) *logical relation* [5, 15, 19], for which a parametricity theorem can

[2] By contrast, if, e.g., recursion were performed at a type whose support for it relied on pointedness of α, then even in the absence of *seq* we could conclude *filter*'s free theorem only for strict h.

be proved. (See Section 4 for details.) A parametricity theorem asserts that every closed term satisfies the parametricity property derived from its type, i.e., is related to itself by the interpretation of its type. Proofs of such theorems proceed by induction on the syntactic structure of terms, driven by type assignment rules, where the constants occurring in a language form base cases.

Since, in the relational interpretation of a polymorphic type, bound type variables can be interpreted by arbitrary relations (or by arbitrary strict and continuous relations if the language supports pattern matching and fixpoints), the parametricity property for each term of polymorphic type is quite general. This allows a richness of free theorems to be derived by instantiation. But this generality is also the potential downfall of free theorems when *seq* is added to the language as a constant. The proof of the associated base case fails because, in the setting of the standard relational interpretations, *seq* does not satisfy the parametricity property derived from its type: the encountered relations are too unconstrained since they allow $\perp$ and non-$\perp$ values to be related arbitrarily. (See Section 5 for why exactly this is problematic.)

This observation motivates us to adapt the standard construction of a logical relation. As expected, bound type variables are only allowed to be interpreted by relations that validate the property derived from the polymorphic type which *seq* claims for itself. But the relational interpretation of function types must also be modified. Indeed, it must conform to a notion of extensionality which acknowledges that, in the presence of *seq*, two functions can no longer be considered semantically equivalent simply because they map the same arguments to the same results. The need to alter the construction of the logical relation at function types is perhaps surprising, given that the conventional wisdom has not anticipated it.

Of the different options for restricting relations to ensure the adherence of *seq* to its parametricity property, we choose one that introduces a certain asymmetry into the logical relation. Since equations proved as free theorems in the absence of *seq* are frequently disproved in its presence by instantiations that make one of their two sides less defined than (but not completely unrelated to) the other, we define our new logical relation in such a way that one of the two argument positions of a relational interpretation is "favored" with respect to definedness. This leads us to interpret base types as semantic approximation relations rather than as identity relations. Further, we modify the relational interpretations of non-base types by adopting a propagation technique which is more complex than the standard one, but which preserves the chosen set of restrictions.

We sketch the proof of the fundamental property of the new logical relation, establishing that it does indeed relate every closed term (potentially built using *seq*) to itself. We use the fundamental property to prove free theorems in which the modifications to the logical relation mandated by *seq* generate preconditions that must be satisfied by the values over which the theorems are parametrized. This approach allows us, for example, to provide criteria on p and h under which the law (1) holds for every l, even in the presence of *seq*. Moreover, assuming a weaker precondition on h — and placing no restrictions whatsoever on p or l — it yields an inequational version of law (1) in which the right-hand side is at least as defined as the left-hand side. Such inequations are provable for other functions as well, and are often sufficient when applying free theorems to transform or reason about programs.

The remainder of this paper is organized as follows. Section 2 briefly considers the functional language we use. Section 3 introduces auxiliary notions and definitions. Section 4 recalls how free theorems are obtained in the absence of *seq*. Sections 5 and 6 motivate and develop our approach to parametricity in the presence of *seq*. Section 7 applies it to derive two free theorems about functions with *filter*'s type. Section 8 investigates how some program transformations based on free theorems [6, 20, 22] fare in the presence of *seq*. Section 9 concludes by proposing future research directions.

2 Functional Language

We use a subset of the pure nonstrict functional programming language Haskell [12] that corresponds to a Girard-Reynolds-style calculus with fixpoints, algebraic datatypes, and *seq*. Definitions of Haskell types and functions that are used throughout the paper are given in Figure 1. As shown there, parametric polymorphism[3] is made explicit with a $\forall$-type constructor quantifying over types. Type instantiation, on the other hand, is most often left implicit. When instantiation of a polymorphic term t to a *closed type* (one without free variables) τ is made explicit, it is denoted by t_τ.

Unfortunately, there is not yet a formal semantics for Haskell, independent of any concrete implementation. Nevertheless, it is common practice to use a denotational style [18] for reasoning about Haskell programs, and we do so as well. In particular, we use the semantic approximation $\sqsubseteq$ — interpreted as "less than or equally as defined as" — between values of the same type, and the value $\perp$ — interpreted as "undefined" — at every type. Types are taken to be pointed complete partial orders, i.e., sets equipped with the partial order $\sqsubseteq$, the least element $\perp$, and limits of all chains. Programs are taken to be monotonic and continuous functions between pointed complete partial orders. The notions of monotonicity and continuity are given in the next section, together with other preliminaries.

3 Preliminaries

For closed types τ_1 and τ_2 the set of all binary relations between their value sets is denoted by $Rel(\tau_1, \tau_2)$. Functions are special cases of relations, i.e., a function $h :: \tau_1 \to \tau_2$ is interpreted as its graph $\{(x,y) \mid h\,x = y\} \in Rel(\tau_1, \tau_2)$.

For every closed type τ the relations $\sqsubseteq_\tau, \sqsubsetneq_\tau \in Rel(\tau,\tau)$ and the value $\perp_\tau :: \tau$ are the semantic approximation (partial) order, the strict order induced by it, and the least element, respectively, for the interpretation of τ. The subscripts will often be omitted. Nevertheless, the value $\perp :: \tau$ at a particular closed type should not be confused with the polymorphic value $\perp :: \forall \alpha. \alpha$.

Let τ_1 be a type with at most one free variable, say α. For every closed type τ, $\tau_1[\tau/\alpha]$ denotes the result of substituting τ for all free occurrences of α in τ_1. For every value $u :: \forall \alpha. \tau_1$ we have:

$$\boxed{(\forall \tau.\ u_\tau = \perp_{\tau_1[\tau/\alpha]}) \Rightarrow u = \perp_{\forall \alpha. \tau_1} \qquad (2)}$$

A relation is *strict* if it contains the pair $(\perp, \perp)$. A relation is *total* if, for every pair (x,y) contained in it, $x \neq \perp$ implies $y \neq \perp$. A relation is *bottom-reflecting* if, for every pair (x,y) contained in it, $x \neq \perp$ iff $y \neq \perp$. A relation is *continuous* if the limits of two chains of pairwise related elements are again related. A relation is *admissible* if it is strict and continuous. A function h is *monotonic* if $x \sqsubseteq y$ implies $h\,x \sqsubseteq h\,y$.

[3] The ad-hoc polymorphism also provided by Haskell in the form of type classes [24] is not considered here. How to handle type classes when deriving free theorems is discussed briefly in [23].

The composition of two relations $\mathcal{R} \in Rel(\tau_1,\tau_2)$ and $\mathcal{S} \in Rel(\tau_2,\tau_3)$ is defined as:

$$\mathcal{R};\mathcal{S} = \{(x,z) \mid \exists y.\ (x,y) \in \mathcal{R} \wedge (y,z) \in \mathcal{S}\} \in Rel(\tau_1,\tau_3).$$

A relation $\mathcal{R}$ is *left-closed* if $\sqsubseteq;\mathcal{R} = \mathcal{R}$. The inverse of a relation $\mathcal{R} \in Rel(\tau_1,\tau_2)$ is defined as:

$$\mathcal{R}^{-1} = \{(y,x) \mid (x,y) \in \mathcal{R}\} \in Rel(\tau_2,\tau_1).$$

We denote $\sqsubseteq^{-1}$ and $\sqsubsetneq^{-1}$ by $\sqsupseteq$ and $\sqsupsetneq$, respectively.

The lifting of a relation $\mathcal{R} \in Rel(\tau_1,\tau_2)$ to Maybe types, $lift_{\mathsf{Maybe}}(\mathcal{R}) \in Rel(\mathsf{Maybe}\ \tau_1, \mathsf{Maybe}\ \tau_2)$, is defined as:

$$\{(\bot,\bot), (\mathsf{Nothing},\mathsf{Nothing})\} \cup \{(\mathsf{Just}\ x, \mathsf{Just}\ y) \mid (x,y) \in \mathcal{R}\}.$$

If $\mathcal{R}$ is continuous, then so is $lift_{\mathsf{Maybe}}(\mathcal{R})$. If $\mathcal{R}$ is left-closed in addition, then $\sqsubseteq;lift_{\mathsf{Maybe}}(\mathcal{R})$ is also continuous.

The lifting of relations $\mathcal{R} \in Rel(\tau_1,\tau_2)$ and $\mathcal{S} \in Rel(\tau'_1,\tau'_2)$ to pairs, $lift_{(,)}(\mathcal{R},\mathcal{S}) \in Rel((\tau_1,\tau'_1),(\tau_2,\tau'_2))$, is defined as:

$$\{(\bot,\bot)\} \cup \{((x,x'),(y,y')) \mid (x,y) \in \mathcal{R} \wedge (x',y') \in \mathcal{S}\}.$$

If $\mathcal{R}$ and $\mathcal{S}$ are continuous, then so is $lift_{(,)}(\mathcal{R},\mathcal{S})$. If $\mathcal{R}$ and $\mathcal{S}$ are left-closed in addition, then $\sqsubseteq;lift_{(,)}(\mathcal{R},\mathcal{S})$ is also continuous.

The lifting of a relation $\mathcal{R} \in Rel(\tau_1,\tau_2)$ to lists, $lift_{[]}(\mathcal{R}) \in Rel([\tau_1],[\tau_2])$, is defined as the largest $\mathcal{S} \in Rel([\tau_1],[\tau_2])$ such that:

$$\mathcal{S} = \{(\bot,\bot), ([],[])\} \cup \{(x:xs, y:ys) \mid (x,y) \in \mathcal{R} \wedge (xs,ys) \in \mathcal{S}\}.$$

Thus, two lists are related by $lift_{[]}(\mathcal{R})$ if (i) either both are finite or partial lists of same length, or both are infinite lists, and (ii) elements at corresponding positions are related by $\mathcal{R}$. If $\mathcal{R}$ is continuous, then so is $lift_{[]}(\mathcal{R})$. If $\mathcal{R}$ is left-closed in addition, then $\sqsubseteq;lift_{[]}(\mathcal{R})$ is also continuous. Note that in the special case that $\mathcal{R}$ is the graph of a Haskell function h, the relation $lift_{[]}(\mathcal{R})$ coincides with the graph of the function *map h* as defined in Figure 1.

Since $\sqsubseteq$ is reflexive, it is not hard to see from the above definitions that for all appropriately typed functions h and lists l:

$$(map\ h\ l, l) \in \sqsubseteq;lift_{[]}(\sqsubseteq;h^{-1}) \tag{3}$$

$$(l, map\ h\ l) \in \sqsubseteq;lift_{[]}(h;\sqsubseteq) \tag{4}$$

Moreover, for all functions h and all appropriately typed lists l_1 and l_2 we can conclude from transitivity of $\sqsubseteq$ and the obvious inclusion of $lift_{[]}(\sqsubseteq;h^{-1})$ in $\sqsubseteq;(map\ h)^{-1}$ that:

$$(l_1,l_2) \in \sqsubseteq;lift_{[]}(\sqsubseteq;h^{-1}) \Rightarrow l_1 \sqsubseteq map\ h\ l_2 \tag{5}$$

If h is monotonic, then from monotonicity of *map h* and the inclusion of $lift_{[]}(h;\sqsubseteq)$ in $(map\ h);\sqsubseteq$ we similarly obtain:

$$(l_2,l_1) \in \sqsubseteq;lift_{[]}(h;\sqsubseteq) \Rightarrow map\ h\ l_2 \sqsubseteq l_1 \tag{6}$$

4 Free Theorems in the Absence of *seq*

As discussed in the introduction, the key to deriving free theorems from types is to interpret types as relations. Each type variable is thus interpreted as a relation, and associated with every type constructor of the calculus is a map which produces a new relational interpretation from an appropriate number of given ones. Such a map is called a *relational action*. We first recall the standard relational actions for the type constructors of the Girard-Reynolds polymorphic lambda calculus.

The relational action corresponding to the function type constructor maps two relations $\mathcal{R} \in Rel(\tau_1,\tau_2)$ and $\mathcal{S} \in Rel(\tau'_1,\tau'_2)$ to the following relation in $Rel(\tau_1 \to \tau'_1, \tau_2 \to \tau'_2)$:

$$\mathcal{R} \to \mathcal{S} = \{(f,g) \mid \forall (x,y) \in \mathcal{R}.\ (f\ x, g\ y) \in \mathcal{S}\}.$$

In other words, two functions are related if they map related arguments to related results.

Let τ_1 and τ_2 be types with at most one free variable, say α, and let $\mathcal{F}$ be a function that, for all closed types τ'_1 and τ'_2 and every relation $\mathcal{R} \in Rel(\tau'_1,\tau'_2)$, gives a relation $\mathcal{F}(\mathcal{R}) \in Rel(\tau_1[\tau'_1/\alpha], \tau_2[\tau'_2/\alpha])$. The relational action corresponding to the $\forall$-type constructor maps $\mathcal{F}$ to the following relation in $Rel(\forall\alpha.\,\tau_1, \forall\alpha.\,\tau_2)$:

$$\begin{aligned}&\forall \mathcal{R} \in Rel.\ \mathcal{F}(\mathcal{R})\\ &= \{(u,v) \mid \forall \tau'_1, \tau'_2, \mathcal{R} \in Rel(\tau'_1,\tau'_2).\ (u_{\tau'_1}, v_{\tau'_2}) \in \mathcal{F}(\mathcal{R})\}.\end{aligned}$$

According to this definition, two polymorphic values are related if all their instances respect the operation of $\mathcal{F}$ on relations between the types at which instantiation occurs.

The relational actions introduced above can be used to define a logical relation by induction on the structure of types. Since we will need to keep track of the interpretations of quantified types, we use *relation environments* to map type variables to relations between closed types. The empty relation environment is denoted by $\emptyset$ and the update, or extension, of a relation environment η by mapping α to $\mathcal{R}$ is denoted by $\eta[\mathcal{R}/\alpha]$.

Let τ be a type and η be a relation environment such that $\eta(\alpha) \in Rel(\tau_{1\alpha},\tau_{2\alpha})$ for each free variable α of τ. We write $\tau_{\overleftarrow{\eta}}$ and $\tau_{\overrightarrow{\eta}}$ for the closed types obtained by replacing every free occurrence of each variable α in τ with $\tau_{1\alpha}$ and $\tau_{2\alpha}$, respectively. A relation $\Delta_{\tau,\eta} \in Rel(\tau_{\overleftarrow{\eta}}, \tau_{\overrightarrow{\eta}})$ is defined as in Figure 2.

$$\begin{aligned}\Delta_{\alpha,\eta} &= \eta(\alpha)\\ \Delta_{\tau\to\tau',\eta} &= \Delta_{\tau,\eta} \to \Delta_{\tau',\eta}\\ \Delta_{\forall\alpha.\,\tau,\eta} &= \forall \mathcal{R} \in Rel.\ \Delta_{\tau,\eta[\mathcal{R}/\alpha]}\end{aligned}$$

Figure 2. Definition of the standard logical relation.

If τ is a closed type, we obtain a relation $\Delta_{\tau,\emptyset} \in Rel(\tau,\tau)$. The *abstraction* or *parametricity theorem* for Δ [17, 23], from which the standard free theorems are derived, then states that for every closed term $t :: \tau$ we have $(t,t) \in \Delta_{\tau,\emptyset}$. This is also called the *fundamental property* of the logical relation.

As outlined so far, the standard logical relation is only defined, and its associated parametricity theorem only holds, for the pure polymorphic lambda calculus. To more closely approximate modern functional languages, we must also take general recursive definitions and suitable datatypes into account. Of course, these features should be added without breaking the fundamental property.

It is well known that the provision of general fixpoint recursion — as is captured by the function *fix* from Figure 1 — requires all relations used in the definition of the logical relation to be admissible. In particular, in the inductive case for the $\forall$-type constructor the quantification of $\mathcal{R}$ must be over admissible relations only. The essential observations are then that admissibility is preserved by the given relational actions, and that it ensures the adherence of *fix* to the parametricity property derived from its type.

When adding base types such as Int or algebraic datatypes such as Bool, Maybe, pairs, and standard Haskell lists, we must define for each new type constructor a corresponding relational action. The usual approach is to interpret nonparametrized datatypes as identity relations and parametrized datatypes by structural liftings of relations. Illustrating examples are given in Figure 3. Further, we must verify that each new constant used to construct or handle values of the new types satisfies the parametricity property derived from its type. For example, we must check that $(i,i) \in \Delta_{\mathsf{Int},\emptyset}$ for every integer literal i, $(+,+) \in \Delta_{\mathsf{Int}\to\mathsf{Int}\to\mathsf{Int},\emptyset}$, $([],[]) \in \Delta_{\forall\alpha.\,[\alpha],\emptyset}$, and $(\mathsf{Just},\mathsf{Just}) \in \Delta_{\forall\alpha.\,\alpha\to\mathsf{Maybe}\,\alpha,\emptyset}$ hold. This is indeed the case, and similarly for other constants.

$$\boxed{\begin{array}{ll}
\Delta_{\mathsf{Int},\eta} & = id_{\mathsf{Int}} \\
\Delta_{\mathsf{Bool},\eta} & = id_{\mathsf{Bool}} \\
\Delta_{\mathsf{Maybe}\,\tau,\eta} & = lift_{\mathsf{Maybe}}(\Delta_{\tau,\eta}) \\
\Delta_{(\tau,\tau'),\eta} & = lift_{(,)}(\Delta_{\tau,\eta},\Delta_{\tau',\eta}) \\
\Delta_{[\tau],\eta} & = lift_{[]}(\Delta_{\tau,\eta})
\end{array}}$$

Figure 3. Standard relational interpretations for datatypes.

For each algebraic datatype we must also include a means of destructing its values via pattern matching. One way to do this is to introduce, as a new term-forming operation, a **case**-construct that can be used at every algebraic datatype.[4] Proving that the fundamental property of the logical relation remains intact amounts to simply checking that the relational interpretation of every return type of a **case**-expression is strict. That strictness is all that is required can be argued as follows.

By analogy with Reynolds' abstraction theorem we must show that two denotations of a **case**-expression in related environments are related by the interpretation of its type whenever it is constructed from subterms whose denotations are similarly related. To this end, we first note that the relational interpretation of an algebraic datatype respects its structure. Thus, if two denotations of the selector of a **case**-expression are related by the interpretation of the selector's (algebraic) datatype, then the structural nature of this interpretation ensures that pattern matching against either denotation will select the same branch, if any, of the **case**-expression. There are two cases to consider. If pattern matching against one — and hence both — selector denotations succeeds, then, by hypothesis, the resulting denotations of the **case**-expression are values that are related by the interpretation of its return type. If, on the other hand, pattern matching fails on one of the selector denotations (either because the value is $\bot$ or because the pattern match is not exhaustive), then it also fails on the other one. What we need to establish then is that the interpretation determined by the logical relation for the return type of the **case**-expression relates the two resulting $\bot$s. It clearly does if it is known to be strict. This strictness requirement is also reflected in the pointedness constraint on the result type of the prototypical **case**-constant given in Section 3 of [11].

Like the relational actions for the function and $\forall$-type constructors, the relational actions for algebraic datatypes as used in Figure 3 preserve admissibility. In particular, all relational actions preserve strictness, as is required for **case**-expressions to satisfy their parametricity properties. The restricted quantification over admissible relations in the $\forall$-case thus ensures that the resulting free theorems are valid for programs potentially using both general recursion and the richer type structure. In the next section we turn to the question of what happens when *seq* is also added to the language.

5 Free Theorems Fail in the Presence of *seq*

When adding the constant *seq* to the language, we must ensure that it satisfies the parametricity property derived from its type:

$$\begin{array}{ll}
& (seq,seq) \in \Delta_{\forall\alpha\beta.\,\alpha\to\beta\to\beta,\emptyset} \\
\Leftrightarrow & (seq,seq) \in (\forall \mathcal{R}\in Rel.\,\Delta_{\forall\beta.\,\alpha\to\beta\to\beta,[\mathcal{R}/\alpha]}) \\
\Leftrightarrow & (seq,seq) \in (\forall \mathcal{R}\in Rel.\,\forall \mathcal{S}\in Rel.\,\Delta_{\alpha\to\beta\to\beta,[\mathcal{R}/\alpha,\mathcal{S}/\beta]}) \\
\Leftrightarrow & (seq,seq) \in (\forall \mathcal{R}\in Rel.\,\forall \mathcal{S}\in Rel.\,\mathcal{R}\to(\mathcal{S}\to\mathcal{S})) \\
\Leftrightarrow & \forall \mathcal{R}\in Rel(\tau_1,\tau_2).\ (seq_{\tau_1},seq_{\tau_2}) \in (\forall \mathcal{S}\in Rel.\,\mathcal{R}\to(\mathcal{S}\to\mathcal{S})) \\
\Leftrightarrow & \forall \mathcal{R}\in Rel(\tau_1,\tau_2), \mathcal{S}\in Rel(\tau'_1,\tau'_2). \\
& \quad (seq_{\tau_1\,\tau'_1},seq_{\tau_2\,\tau'_2}) \in \mathcal{R}\to(\mathcal{S}\to\mathcal{S}) \\
\Leftrightarrow & \forall \mathcal{R}\in Rel(\tau_1,\tau_2), \mathcal{S}\in Rel(\tau'_1,\tau'_2), (a_1,a_2)\in\mathcal{R}. \\
& \quad (seq_{\tau_1\,\tau'_1}\ a_1, seq_{\tau_2\,\tau'_2}\ a_2) \in \mathcal{S}\to\mathcal{S} \\
\Leftrightarrow & \forall \mathcal{R}\in Rel(\tau_1,\tau_2), \mathcal{S}\in Rel(\tau'_1,\tau'_2), (a_1,a_2)\in\mathcal{R}, (b_1,b_2)\in\mathcal{S}. \\
& \quad (seq_{\tau_1\,\tau'_1}\ a_1\ b_1, seq_{\tau_2\,\tau'_2}\ a_2\ b_2) \in \mathcal{S}.
\end{array}$$

But even if we restrict ourselves to admissible relations, the resulting statement is not true. As a counterexample, consider the following instantiation:

$$\begin{array}{lll}
\mathcal{R} & = \bot_{\mathsf{Bool}\to\mathsf{Bool}} & \in Rel(\mathsf{Bool},\mathsf{Bool}) \\
\mathcal{S} & = id_{\mathsf{Bool}} & \in Rel(\mathsf{Bool},\mathsf{Bool}) \\
(a_1,a_2) & = (\mathsf{False},\bot_{\mathsf{Bool}}) & \in \mathcal{R} \\
(b_1,b_2) & = (\mathsf{False},\mathsf{False}) & \in \mathcal{S}.
\end{array}$$

Although $\mathcal{R}$ and $\mathcal{S}$ are admissible, the claimed membership $(seq\ \mathsf{False}\ \mathsf{False}, seq\ \bot_{\mathsf{Bool}}\ \mathsf{False}) \in \mathcal{S}$ does not hold.

Since *seq* violates the parametricity property dictated by its type, other terms that are built using *seq* might do so as well. The problem lies with relations, such as $\mathcal{R}$ above, that relate $\bot$ to non-$\bot$ values. It has therefore been proposed (e.g., in [11]) that quantified relations should be further restricted by requiring bottom-reflectingness. But the first counterexample in the introduction shows that, contrary to conventional wisdom, this is not quite enough to recover valid free theorems in the presence of *seq*. (The requirement that relations be admissible and bottom-reflecting becomes a strictness and totality requirement on Haskell functions which instantiate those relations.) The catch is that we must not only impose appropriate restrictions on the quantified relations, but must also ensure that these restrictions are preserved by all relational actions. This is crucial because during the proof of the parametricity theorem, in the inductive case for type instantiation, a universally quantified relation that is subject to the imposed restrictions is instantiated with the relational interpretation of an arbitrary type to establish the induction conclusion. If that interpretation is not guaranteed to fulfill the necessary restrictions, then this use of the induction hypothesis is impossible, and the entire proof breaks. The proof of the parametricity theorem breaks in precisely this way if one subjects quantified relations to bottom-reflectingness but sticks to the standard relational action for the function type constructor. To see why, consider that for every relation $\mathcal{R}$ and every strict relation $\mathcal{S}$, the relation $\mathcal{R}\to\mathcal{S}$ contains the pair $(\bot,(\lambda x\to\bot))$ and consequently — since $(\lambda x\to\bot)$ is different from $\bot$ in the presence of *seq* — is not bottom-reflecting as would be required.

In the next section we solve this problem by modifying the action of $\to$ on relations to take into account the difference between an

[4] Other uses of pattern matching in Haskell programs — e.g., on left-hand sides of function equations — can all be translated into **case**-expressions.

undefined function and a defined function that always returns an undefined value. This is done by explicitly adding a condition on the definedness of related functions similar to that in the *lazy logical relation* (in the absence of polymorphism and algebraic datatypes) of [4]. But rather than requiring bottom-reflectingness, we impose other restrictions on relational interpretations. In contrast to only recovering the usual equational free theorems under quite severe preconditions, these restrictions additionally allow us to derive inequational versions of these theorems under weaker preconditions.

6 Recovering Free Theorems in the Presence of *seq*

As demonstrated in the previous section, the presence of *seq* causes problems if relational interpretations of types are allowed to relate $\bot$ and non-$\bot$ values. One way to accomodate *seq* would thus be to require all encountered relations (those used to interpret bound type variables and those obtained via the relational actions) to be bottom-reflecting in addition to being admissible. This is a drastic restriction, however, because it entails that the statements that are finally obtained as free theorems will only capture situations in which either both of the sides of a law are undefined or neither is. But as we have seen in the introduction, the very nature of *seq*'s impact on free theorems provable in its absence is to potentially make one of the two sides of a law less defined (indeed, potentially $\bot$), while the other one remains unchanged. To derive interesting statements for such situations, we should therefore be more liberal.

We introduce an asymmetry into relational interpretations by allowing relations $\mathcal{R}$ surfacing in the new logical relation to contain pairs $(\bot, y)$ with $y \neq \bot$, but forbidding pairs $(x, \bot)$ with $x \neq \bot$. Thus, instead of requiring totality of both $\mathcal{R}$ and $\mathcal{R}^{-1}$ — which amounts to bottom-reflectingness — we require only totality of $\mathcal{R}$. One set of restrictions that encompasses this asymmetry idea and ensures the adherence of *seq* to the parametricity property derived from its type was proposed in Appendix B of [22]. However, the same lapse occurred there as in the "conventional wisdom" mentioned in the previous section: that those restrictions are preserved by all relational actions was never verified. Unfortunately, they are not so preserved, and their ad-hoc nature makes it uncertain whether they would provide a good starting point for putting things right by adjusting the relational actions.

It is our desire to account for the fact that one of the two terms related by a free theorem can become strictly less defined than the other in the presence of *seq*. This motivates turning our attention to relations which are left-closed in addition to being admissible and total. The need to ensure that all relational actions preserve these restrictions forces us to adjust their definitions. But preserving admissibility, totality, and left-closedness is not the only concern when adjusting the definitions of the relational actions. The new relational actions must also lend themselves to proving a parametricity theorem, and must therefore have an "extensional flavor" which ties them to the semantics of the language.[5] Extensionality here is with respect to semantic approximation. For example, the new relational action for the function type constructor applied to relations $\sqsubseteq_\tau$ and $\sqsubseteq_{\tau'}$ will capture exactly the conditions under which a function $f :: \tau \to \tau'$ approximates a function $g :: \tau \to \tau'$ in the presence of *seq*.

In the remainder of this paper the set of all admissible, total, and left-closed relations between values of closed types τ_1 and τ_2 is denoted by $Rel^{seq}(\tau_1, \tau_2)$. The relational action corresponding to the function type constructor is adapted to map relations $\mathcal{R} \in Rel^{seq}(\tau_1, \tau_2)$ and $\mathcal{S} \in Rel^{seq}(\tau'_1, \tau'_2)$ to the following relation in $Rel^{seq}(\tau_1 \to \tau'_1, \tau_2 \to \tau'_2)$:

$$\begin{aligned}\mathcal{R} \to^{seq} \mathcal{S} = \{(f,g) \mid \; & (f \neq \bot \Rightarrow g \neq \bot) \\ & \wedge \forall (x,y) \in \mathcal{R}.\ (f\ x, g\ y) \in \mathcal{S}\},\end{aligned}$$

i.e., we explicitly add the totality restriction. That the resulting relation is admissible follows from monotonicity and continuity of function application in Haskell and from admissibility of $\mathcal{S}$. To show that it is also left-closed, we need to establish that from $f' \sqsubseteq f$ and $(f,g) \in \mathcal{R} \to^{seq} \mathcal{S}$ it follows that $(f',g) \in \mathcal{R} \to^{seq} \mathcal{S}$, i.e.,

$$(f' \neq \bot \Rightarrow g \neq \bot) \;\wedge\; \forall (x,y) \in \mathcal{R}.\ (f'\ x, g\ y) \in \mathcal{S}.$$

The first conjunct follows from $f' \sqsubseteq f$ and $f \neq \bot \Rightarrow g \neq \bot$. The second conjunct follows by left-closedness of $\mathcal{S}$ from the facts that, for every $(x,y) \in \mathcal{R}$, we have $f'\ x \sqsubseteq f\ x$ by monotonicity of function application in Haskell, as well as $(f\ x, g\ y) \in \mathcal{S}$.

The relational action corresponding to the $\forall$-type constructor is adapted by quantifying only over admissible, total, and left-closed relations as follows. Let τ_1 and τ_2 be types with at most one free variable, say α. In addition, let $\mathcal{F}$ be a function that, for all closed types τ'_1 and τ'_2 and every relation $\mathcal{R} \in Rel^{seq}(\tau'_1, \tau'_2)$, gives a relation $\mathcal{F}(\mathcal{R}) \in Rel^{seq}(\tau_1[\tau'_1/\alpha], \tau_2[\tau'_2/\alpha])$. The new relational action maps $\mathcal{F}$ to the following relation in $Rel^{seq}(\forall\alpha.\,\tau_1, \forall\alpha.\,\tau_2)$:

$$\begin{aligned}&\forall \mathcal{R} \in Rel^{seq}.\ \mathcal{F}(\mathcal{R}) \\ &= \{(u,v) \mid \forall \tau'_1, \tau'_2, \mathcal{R} \in Rel^{seq}(\tau'_1, \tau'_2).\ (u_{\tau'_1}, v_{\tau'_2}) \in \mathcal{F}(\mathcal{R})\}.\end{aligned}$$

That the resulting relation is admissible follows from monotonicity and continuity of type instantiation in Haskell and from admissibility of all the $\mathcal{F}(\mathcal{R})$. That it is total can be shown by indirect reasoning as follows. Assume $(u, \bot_{\forall\alpha.\,\tau_2}) \in (\forall \mathcal{R} \in Rel^{seq}.\ \mathcal{F}(\mathcal{R}))$ for some $u \neq \bot_{\forall\alpha.\,\tau_1}$. Since for every closed type τ we have $\sqsubseteq_\tau \in Rel^{seq}(\tau, \tau)$, this means that for every such τ we have $(u_\tau, (\bot_{\forall\alpha.\,\tau_2})_\tau) \in \mathcal{F}(\sqsubseteq_\tau)$, i.e., $(u_\tau, \bot_{\tau_2[\tau/\alpha]}) \in \mathcal{F}(\sqsubseteq_\tau)$. By totality of $\mathcal{F}(\sqsubseteq_\tau)$ it follows that for every such τ we have $u_\tau = \bot_{\tau_1[\tau/\alpha]}$, and thus by law (2) we derive the contradiction that $u = \bot_{\forall\alpha.\,\tau_1}$. To show left-closedness of $\forall \mathcal{R} \in Rel^{seq}.\ \mathcal{F}(\mathcal{R})$, we need to establish that from $u' \sqsubseteq u$ and $(u,v) \in (\forall \mathcal{R} \in Rel^{seq}.\ \mathcal{F}(\mathcal{R}))$ it follows that $(u',v) \in (\forall \mathcal{R} \in Rel^{seq}.\ \mathcal{F}(\mathcal{R}))$, i.e.,

$$\forall \tau'_1, \tau'_2, \mathcal{R} \in Rel^{seq}(\tau'_1, \tau'_2).\ (u'_{\tau'_1}, v_{\tau'_2}) \in \mathcal{F}(\mathcal{R}).$$

This follows by left-closedness of all the $\mathcal{F}(\mathcal{R})$ from the facts that, for every τ'_1, τ'_2, and $\mathcal{R} \in Rel^{seq}(\tau'_1, \tau'_2)$, we have $u'_{\tau'_1} \sqsubseteq u_{\tau'_1}$ by monotonicity of type instantiation in Haskell, and $(u_{\tau'_1}, v_{\tau'_2}) \in \mathcal{F}(\mathcal{R})$.

The relational interpretations of datatypes are left-composed with $\sqsubseteq$. It is not hard to see (from the facts that $\sqsubseteq ; \mathcal{R}$ is always strict, total, and left-closed for a strict and total $\mathcal{R}$ and that the standard relational interpretations for datatypes are strict and total by construction) that this gives strict, total, and left-closed relations only. Further, $\sqsubseteq$ is a continuous relation, and for continuous and left-closed relations $\mathcal{R}$ and $\mathcal{S}$ the relations $\sqsubseteq ; lift_{\mathsf{Maybe}}(\mathcal{R})$, $\sqsubseteq ; lift_{(,)}(\mathcal{R}, \mathcal{S})$, and $\sqsubseteq ; lift_{[]}(\mathcal{R})$ are also continuous.

Hence, all relations that turn up in the definition of the new logical relation as given in Figure 4 will in fact be admissible, total, and left-closed. In particular, $\Delta^{seq}_{\tau,\emptyset} \in Rel^{seq}(\tau, \tau)$ for every closed type τ.

[5] In the absence of this consideration, every relational action could simply return the trivial relation $\{(\bot, \bot)\}$. But this is clearly not what we want.

$$\begin{aligned}
\Delta^{seq}_{\alpha,\eta} &= \eta(\alpha)\\
\Delta^{seq}_{\tau\to\tau',\eta} &= \Delta^{seq}_{\tau,\eta} \to^{seq} \Delta^{seq}_{\tau',\eta}\\
\Delta^{seq}_{\forall\alpha.\,\tau,\eta} &= \forall\mathcal{R}\in Rel^{seq}.\ \Delta^{seq}_{\tau,\eta[\mathcal{R}/\alpha]}\\
\Delta^{seq}_{\mathsf{Int},\eta} &= \sqsubseteq_{\mathsf{Int}}\\
\Delta^{seq}_{\mathsf{Bool},\eta} &= \sqsubseteq_{\mathsf{Bool}}\\
\Delta^{seq}_{\mathsf{Maybe}\,\tau,\eta} &= \sqsubseteq ; lift_{\mathsf{Maybe}}(\Delta^{seq}_{\tau,\eta})\\
\Delta^{seq}_{(\tau,\tau'),\eta} &= \sqsubseteq ; lift_{(,)}(\Delta^{seq}_{\tau,\eta},\Delta^{seq}_{\tau',\eta})\\
\Delta^{seq}_{[\tau],\eta} &= \sqsubseteq ; lift_{[]}(\Delta^{seq}_{\tau,\eta})
\end{aligned}$$

Figure 4. Definition of the logical relation in the presence of *seq*.

We claim that our changed logical relation still has the following fundamental property (from which the new free theorems will be derived):

if τ is a closed type and $t :: \tau$ is a closed term, then:

$$(t,t) \in \Delta^{seq}_{\tau,\emptyset} \qquad (7)$$

The first thing to check is that this is true for *seq*:

$$\begin{aligned}
&\quad (seq, seq) \in \Delta^{seq}_{\forall\alpha\beta.\,\alpha\to\beta\to\beta,\emptyset}\\
\Leftrightarrow\ & \forall\mathcal{R}\in Rel^{seq}(\tau_1,\tau_2), \mathcal{S}\in Rel^{seq}(\tau_1',\tau_2').\\
&\quad (seq_{\tau_1\tau_1'}, seq_{\tau_2\tau_2'}) \in \mathcal{R} \to^{seq} (\mathcal{S} \to^{seq} \mathcal{S})\\
\Leftrightarrow\ & \forall\mathcal{R}\in Rel^{seq}(\tau_1,\tau_2), \mathcal{S}\in Rel^{seq}(\tau_1',\tau_2').\\
&\qquad (seq_{\tau_1\tau_1'} \neq \bot \Rightarrow seq_{\tau_2\tau_2'} \neq \bot)\\
&\quad \wedge \forall(a_1,a_2)\in\mathcal{R}.\ (seq_{\tau_1\tau_1'}\ a_1, seq_{\tau_2\tau_2'}\ a_2) \in \mathcal{S} \to^{seq} \mathcal{S}\\
\Leftrightarrow\ & \forall\mathcal{R}\in Rel^{seq}(\tau_1,\tau_2), \mathcal{S}\in Rel^{seq}(\tau_1',\tau_2').\\
&\qquad (seq_{\tau_1\tau_1'} \neq \bot \Rightarrow seq_{\tau_2\tau_2'} \neq \bot)\\
&\quad \wedge \forall(a_1,a_2)\in\mathcal{R}.\\
&\qquad\quad (seq_{\tau_1\tau_1'}\ a_1 \neq \bot \Rightarrow seq_{\tau_2\tau_2'}\ a_2 \neq \bot)\\
&\qquad \wedge \forall(b_1,b_2)\in\mathcal{S}.\,(seq_{\tau_1\tau_1'}\ a_1\ b_1, seq_{\tau_2\tau_2'}\ a_2\ b_2) \in \mathcal{S}.
\end{aligned}$$

The two implications arising from totality can be discharged because both $seq_{\tau_2\tau_2'}$ and $seq_{\tau_2\tau_2'}\ a_2$ are only partially applied and hence are weak head normal forms different from $\bot$. The statement $(seq\ a_1\ b_1, seq\ a_2\ b_2) \in \mathcal{S}$ under the assumptions $(a_1,a_2) \in \mathcal{R}$ and $(b_1,b_2) \in \mathcal{S}$ is verified by case distinction on a_1 and a_2:

$$\begin{aligned}
a_1 \neq \bot \wedge a_2 \neq \bot &\quad\Rightarrow\quad (seq\ a_1\ b_1, seq\ a_2\ b_2) = (b_1,b_2)\\
a_1 = \bot \wedge a_2 \neq \bot &\quad\Rightarrow\quad (seq\ a_1\ b_1, seq\ a_2\ b_2) = (\bot,b_2)\\
a_1 = \bot \wedge a_2 = \bot &\quad\Rightarrow\quad (seq\ a_1\ b_1, seq\ a_2\ b_2) = (\bot,\bot)
\end{aligned}$$

The case $a_1 \neq \bot$ and $a_2 = \bot$ cannot occur due to totality of $\mathcal{R}$. In the other cases, $(seq\ a_1\ b_1, seq\ a_2\ b_2) \in \mathcal{S}$ follows from $(b_1,b_2) \in \mathcal{S}$ and left-closedness and strictness of $\mathcal{S}$.

We also need to establish that each constant associated with a datatype fulfills the parametricity property derived from its type. For a nonparametrized datatype such as Int this means we must confirm that for every literal $i :: \mathsf{Int}$, $(i,i) \in \Delta^{seq}_{\mathsf{Int},\emptyset} = \sqsubseteq_{\mathsf{Int}}$ holds. This is obviously true, and so are the parametricity properties derived for integer operations such as $+$. For lists, we must confirm that $([],[]) \in \Delta^{seq}_{\forall\alpha.\,[\alpha],\emptyset}$ and $((:),(:)) \in \Delta^{seq}_{\forall\alpha.\,\alpha\to[\alpha]\to[\alpha],\emptyset}$. The latter requires us to establish that certain definedness conditions which arise on partial applications of $(:)$ are satisfied and that for every admissible, total, and left-closed relation $\mathcal{R}$ it follows from $(x,y) \in \mathcal{R}$ and $(xs,ys) \in \sqsubseteq ; lift_{[]}(\mathcal{R})$ that $(x:xs, y:ys) \in \sqsubseteq ; lift_{[]}(\mathcal{R})$. The former is obvious; to prove the latter is an easy exercise using the monotonicity of $(:)$. Similar arguments work for the other data constructors.

What remains to be done is to mirror Wadler's sketched proof [23] that the term-forming operations of the polymorphic lambda calculus — i.e., λ-abstraction, function application, type abstraction, type instantiation — as well as the **case**-construct behave according to the (new) logical relation. As usual, this proof requires a generalization from the statement about closed types τ and closed terms $t :: \tau$ to types and terms potentially containing free variables. It proceeds by induction over the structure of typing derivations.

For the new logical relation, we changed the standard relational action corresponding to the $\forall$-type constructor by imposing admissibility, totality, and left-closedness on the relations over which quantification takes place. This means that, in comparison to the standard proof, the hypothesis in the induction step for the typing rule in whose **premise** a $\forall$-type appears — i.e., the induction hypothesis for the rule for type instantiation — now provides a weaker statement concerning restricted relations only. But since the new relational actions are constructed precisely so that the relational interpretations of all types satisfy the required restrictions, this is just enough to prove the induction conclusion. For the other induction step involving a $\forall$-type — i.e., for the step corresponding to type abstraction — no additional arguments are necessary.

The only change to the relational action corresponding to the function type constructor is a strengthening due to the added totality restriction. Thus, of the two type inference rules involving a function type, only the induction step for the rule in whose **conclusion** the function type appears differs from that for the standard logical relation. For an abstraction of the form $(\lambda x \to t)$ appearing in the conclusion of that rule we must show in addition that $(\lambda x \to [\![t]\!]_{\rho_1}) \neq \bot$ implies $(\lambda x \to [\![t]\!]_{\rho_2}) \neq \bot$ for every pair of type-respecting environments ρ_1 and ρ_2 mapping the free type variables of t to types and the free object variables of t other than x to values. Here $[\![t]\!]_{\rho_1}$ and $[\![t]\!]_{\rho_2}$ denote the values of t in the environments ρ_1 and ρ_2, respectively, where the value of a term in an environment is defined in the usual way. The implication in question obviously holds because λ-abstractions are weak head normal forms distinct from $\bot$.

The induction step for **case**-expressions amounts to considering the different ways in which the denotation, in one environment, of the selector of such an expression can be related to its denotation in a related environment. We must show that for each such possibility the denotations of the whole **case**-expression in the two environments are correspondingly related by the interpretation of its return type. The argument proceeds along the same lines as the corresponding one for the standard logical relation in Section 4. Since each new interpretation of an algebraic datatype is the composition of the semantic approximation ordering and its standard structural interpretation, the argument additionally uses the fact that the relational interpretations of all types are left-closed.

Finally, since admissibility is included among the restrictions we impose on all relations, the arguments from Section 7 of [23] ensure that $\bot$ and *fix* also fulfill their parametricity properties with respect to the new logical relation. Putting everything together, we conclude that (7) holds in the presence of both *seq* and general fixpoint recursion.

6.1 Manufacturing Permissible Relations

An oft-followed strategy when deriving free theorems is to specialize quantified relations to functions. Since the only functions that are strict and left-closed are constant functions mapping to $\bot$, and since such a function is total only when its domain consists solely of $\bot$, this is not very useful in the presence of *seq* and its attendant restrictions on relations. There are, however, two canonical ways to manufacture admissible, total, and left-closed relations out of a function. These are considered now and put to good use in the next two sections.

On the one hand, for every monotonic and admissible function h the relation

$$\sqsubseteq; h^{-1} = \{(x,y) \mid x \sqsubseteq h\ y\}$$

is admissible, total, and left-closed. On the other, for every monotonic, admissible, and total function h the relation

$$h; \sqsubseteq\, = \{(x,y) \mid h\ x \sqsubseteq y\}$$

is admissible, total, and left-closed. Note that the monotonicity and admissibility requirements on h above are essential. But since we will only consider functions definable in Haskell below, and these are always assumed to be monotonic and continuous, we will only explicitly record the strictness precondition in the following.

7 Two Free Theorems about *filter*

In this section we show how the fundamental property of our modified logical relation can be used to derive free theorems in the presence of *seq*.

THEOREM 1. *For every function*

$$\mathit{filter} :: \forall \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

and appropriately typed p, h, and l the following hold:

if h is strict, then:
$$\mathit{filter}\ p\ (\mathit{map}\ h\ l) \sqsubseteq \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l) \tag{8}$$

if $p \neq \bot$ and h is strict and total, then:
$$\mathit{filter}\ p\ (\mathit{map}\ h\ l) = \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l) \tag{9}$$

PROOF. The parametricity property for *filter*'s type is the following instance of law (7):

$$(\mathit{filter}, \mathit{filter}) \in \Delta^{seq}_{\forall \alpha.\,(\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha], \emptyset}.$$

Expanding this statement following the definition from Figure 4 yields that for every choice of closed types τ_1 and τ_2, an admissible, total, and left-closed relation $\mathcal{R} \in \mathit{Rel}^{seq}(\tau_1, \tau_2)$, functions $p_1 :: \tau_1 \to \mathsf{Bool}$ and $p_2 :: \tau_2 \to \mathsf{Bool}$, and lists $l_1 :: [\tau_1]$ and $l_2 :: [\tau_2]$ the following holds:

$$\begin{array}{l} \quad(\mathit{filter}_{\tau_1} \neq \bot \Rightarrow \mathit{filter}_{\tau_2} \neq \bot) \\ \wedge\, ((p_1 \neq \bot \Rightarrow p_2 \neq \bot) \wedge (\forall (x_1, x_2) \in \mathcal{R}.\ p_1\ x_1 \sqsubseteq p_2\ x_2) \\ \quad\Rightarrow \quad (\mathit{filter}_{\tau_1}\ p_1 \neq \bot \Rightarrow \mathit{filter}_{\tau_2}\ p_2 \neq \bot) \\ \qquad\quad \wedge\, ((l_1, l_2) \in \sqsubseteq; \mathit{lift}_{[]}(\mathcal{R}) \\ \qquad\qquad \Rightarrow (\mathit{filter}_{\tau_1}\ p_1\ l_1, \mathit{filter}_{\tau_2}\ p_2\ l_2) \in \sqsubseteq; \mathit{lift}_{[]}(\mathcal{R}))). \end{array}$$

Dropping two conjuncts from the above and strengthening one precondition, we obtain the following weaker statement:

$$\begin{array}{l} p_2 \neq \bot \wedge (\forall (x_1, x_2) \in \mathcal{R}.\ p_1\ x_1 \sqsubseteq p_2\ x_2) \\ \Rightarrow ((l_1, l_2) \in \sqsubseteq; \mathit{lift}_{[]}(\mathcal{R}) \\ \qquad \Rightarrow (\mathit{filter}_{\tau_1}\ p_1\ l_1, \mathit{filter}_{\tau_2}\ p_2\ l_2) \in \sqsubseteq; \mathit{lift}_{[]}(\mathcal{R})). \end{array}$$

We consider two instantiations of this.

First, we instantiate

$$\mathcal{R} = \sqsubseteq; h^{-1}, \quad p_1 = p, \quad p_2 = p \circ h, \quad l_1 = \mathit{map}\ h\ l, \quad l_2 = l$$

for a strict function $h :: \tau_2 \to \tau_1$, giving:

$$\begin{array}{l} p \circ h \neq \bot \wedge (\forall x_1 :: \tau_1, x_2 :: \tau_2.\ x_1 \sqsubseteq h\ x_2 \Rightarrow p\ x_1 \sqsubseteq p\ (h\ x_2)) \\ \Rightarrow ((\mathit{map}\ h\ l, l) \in \sqsubseteq; \mathit{lift}_{[]}(\sqsubseteq; h^{-1}) \\ \qquad \Rightarrow (\mathit{filter}_{\tau_1}\ p\ (\mathit{map}\ h\ l), \mathit{filter}_{\tau_2}\ (p \circ h)\ l) \in \sqsubseteq; \mathit{lift}_{[]}(\sqsubseteq; h^{-1})). \end{array}$$

Since $p \circ h$ has a weak head normal form and hence is not $\bot$, and since the second conjunct of the precondition follows from monotonicity of p, applications of laws (3) and (5) yield (8).

Second, we instantiate

$$\mathcal{R} = h; \sqsubseteq, \quad p_1 = p \circ h, \quad p_2 = p, \quad l_1 = l, \quad l_2 = \mathit{map}\ h\ l$$

for a strict and total function $h :: \tau_1 \to \tau_2$, giving:

$$\begin{array}{l} p \neq \bot \wedge (\forall x_1 :: \tau_1, x_2 :: \tau_2.\ h\ x_1 \sqsubseteq x_2 \Rightarrow p\ (h\ x_1) \sqsubseteq p\ x_2) \\ \Rightarrow ((l, \mathit{map}\ h\ l) \in \sqsubseteq; \mathit{lift}_{[]}(h; \sqsubseteq) \\ \qquad \Rightarrow (\mathit{filter}_{\tau_1}\ (p \circ h)\ l, \mathit{filter}_{\tau_2}\ p\ (\mathit{map}\ h\ l)) \in \sqsubseteq; \mathit{lift}_{[]}(h; \sqsubseteq)). \end{array}$$

Since the second conjunct of the precondition follows from monotonicity of p, applications of laws (4) and (6) yield

$$p \neq \bot \Rightarrow \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l) \sqsubseteq \mathit{filter}\ p\ (\mathit{map}\ h\ l),$$

which together with the previously proven (8) gives (9). □

To illustrate the roles of the restrictions on p and h required in the previous theorem, we consider the instantiations for p, h, and l that were used in the introduction — together with the particular function definition for *filter* presented there — as counterexamples for the unrestricted equational law (1). While the first two of these instantiations satisfy law (8), which states that strictness of h is sufficient to guarantee that the right-hand side is at least as defined as the left-hand side, the fourth instantiation demonstrates that strictness of h is not a necessary condition for this. On the other hand, the third instantiation shows that a proof of law (8) without the strictness of h cannot exist. Further, the first two instantiations show that neither the restriction that $p \neq \bot$ nor totality of h can be omitted when recovering the equality in law (9).

Another illustrative take on laws (8) and (9) is to argue on an intuitive level why they hold for the particular function definition of *filter* from the introduction. We consider only the impact of *seq*, as opposed to why law (1) would hold in the first place, i.e., assuming all invocations of *seq* were dropped. First, for law (8), we need to establish that $\mathit{rhs} = \mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ l)$ is always at least as defined as $\mathit{lhs} = \mathit{filter}\ p\ (\mathit{map}\ h\ l)$ for strict h. To do so, we consider all uses of *seq* in the definition of *filter*. The one on *filter*'s first argument turns *lhs* into $\bot$ if $p = \bot$, but never has an impact on *rhs* because $p \circ h$ is always different from $\bot$. If the application of *seq* on some list element x of l finds a $\bot$ in *rhs*, then by strictness of h the corresponding element in $\mathit{map}\ h\ l$ is also $\bot$, and hence the corresponding application of *seq* in *lhs* has the same outcome. A similar observation holds for the application of *seq* on some tail *xs* of l because $\mathit{map}\ h\ \bot = \bot$.

Turning to law (9), we must argue that under the additional restrictions $p \neq \bot$ and totality of h, *lhs* is also at least as defined as *rhs*. This argument breaks naturally into three parts. First note that the new condition on p guarantees that the application of *seq* on p does not result in *lhs* being $\bot$. Second, totality of h guarantees that the application of *seq* to some list element of $\mathit{map}\ h\ l$ in *lhs* only encounters $\bot$ if the corresponding list element of l in *rhs* is itself $\bot$. Strictness of $\mathit{map}\ h$ thus ensures that *rhs* is never any more defined than *lhs* as a result of an application of *seq* to an element of $\mathit{map}\ h\ l$. Finally, the application of *seq* on *xs* leads to no difference between *lhs* and *rhs* because *map h* is total in addition to being strict, and because applying the strict function *h* to all list elements necessarily preserves any resulting undefined list element.

Note that Theorem 1 is really much more general than described in the above discussions because it holds for **every** function *filter* of appropriate type and does not require any knowledge of the concrete function definition. This wide applicability is what has earned free theorems their name. They are now restored to their former glory, even in the presence of *seq*. The proof of the inequational free theorem (8) was made possible by the asymmetry built into the new logical relation. In particular, if we were to replace the totality and left-closedness requirements on relational interpretatations with bottom-reflectingness, and if we were to properly construct a logical relation that preserves these restrictions (which can be done), then we would only be able to obtain law (9).

8 Program Transformations

Free theorems have found an important application as justifications for various kinds of program transformations for nonstrict functional languages [3, 6, 9, 20, 22]. The presence of *seq*, however, threatens the claimed semantics-preserving character of such transformations. Fortunately, one often needs to know only that a program resulting from a transformation is at least as defined as the program from which it was obtained. In such situations the inequational free theorems derived from our new asymmetric logical relation and its associated parametricity theorem come to the rescue. In this section we apply them to evaluate the effect of *seq* on program transformations founded on the polymorphic types of arguments to the functions *destroy*, *build*, and $vanish_{++}$ given in Figure 5.

```
unfoldr :: ∀α β. (β → Maybe (α, β)) → β → [α]
unfoldr  f b = case  f b of Nothing    → []
                            Just (a, b′) → a : unfoldr  f b′

destroy :: ∀α γ. (∀β. (β → Maybe (α, β)) → β → γ)
                  → [α] → γ
destroy g = g listpsi where listpsi []       = Nothing
                            listpsi (a : as) = Just (a, as)

foldr :: ∀α β. (α → β → β) → β → [α] → β
foldr c n []       = n
foldr c n (a : as) = c a (foldr c n as)

build :: ∀α. (∀β. (α → β → β) → β → β) → [α]
build g = g (:) []

vanish₊₊ :: ∀α. (∀β. β → (α → β → β)
                     → (β → β → β) → β) → [α]
vanish₊₊ g = g id (λx h ys → x : h ys) (∘) []
```

Figure 5. Functions for program transformations.

8.1 The Dual of Short Cut Fusion

Svenningsson [20] considered the *destroy*/*unfoldr* rule as a dual to the *foldr*/*build* rule used in short cut fusion [6] (considered in the next subsection). It solves some problems that short cut fusion has with list consumption by *zip*-like functions and by functions defined using accumulating parameters. The *destroy*/*unfoldr* rule can be used to eliminate intermediate lists in compositions of list producers written with *unfoldr* and list consumers written with *destroy*. Svenningsson describes the rule as an oriented replacement transformation, but makes no precise statement about its semantics. He only suggests that correctness of the transformation might be provable using a free theorem. In order for the rule to be safely applicable — i.e., to produce a program that is at least as defined as the original one — we must at least have the following for appropriately typed *g*, *psi*, and *e*:[6]

$$\boxed{destroy\ g\ (unfoldr\ psi\ e) \sqsubseteq g\ psi\ e \qquad (10)}$$

While [20] proposes the *destroy*/*unfoldr* rule for the language Haskell and even contains an example involving *seq*, the possible impact of *seq* on the correctness of the transformation is ignored. But the following two instantiations using *seq* break conjecture (10), making the right-hand side less defined than the left-hand side:

$$\begin{array}{lll} g = (\lambda x\, y \to seq\ x\ []) & psi = \bot & e = [] \\ g = (\lambda x\, y \to seq\ y\ []) & psi = (\lambda x \to \mathsf{Nothing}) & e = \bot \end{array}$$

Thus, in the presence of *seq* the transformation is unsafe.

To find conditions under which (10) holds and (potentially different) conditions under which the converse inequation holds (which together would give conditions for semantic equivalence), we derive the parametricity property for terms of *g*'s type in the definition of *destroy* and instantiate it in such a way that the result relates the two sides of the *destroy*/*unfoldr* rule. While doing so, we keep track of conditions to impose so that the chosen instantiation is permissible (cf. Section 6.1). This process does not immediately yield the inequations we seek, but instead gives free theorems relating the two sides of the *destroy*/*unfoldr* rule by the interpretation of *g*'s return type according to our logical relation. The inequations are then obtained from these theorems under a certain (reasonable) assumption about the interpretations of closed types according to the logical relation, to be discussed below.

THEOREM 2. *For all closed types τ and τ', every function*

$$g :: \forall\beta.\ (\beta \to \mathsf{Maybe}\ (\tau', \beta)) \to \beta \to \tau,$$

and appropriately typed psi and e the following hold:

$$\boxed{\begin{array}{l} \textit{if } psi \neq \bot \textit{ and } psi \textit{ is strict, then:} \\ \qquad (destroy\ g\ (unfoldr\ psi\ e), g\ psi\ e) \in \Delta^{seq}_{\tau,\emptyset} \qquad (11) \end{array}}$$

$$\boxed{\begin{array}{l} \textit{if } psi \textit{ is strict and total and never returns } \mathsf{Just}\ \bot\textit{, then:} \\ \qquad (destroy\ g\ (unfoldr\ psi\ e), g\ psi\ e) \in (\Delta^{seq}_{\tau,\emptyset})^{-1} \qquad (12) \end{array}}$$

PROOF. The parametricity property associated with *g*'s type is the following instance of law (7):

$$(g, g) \in \Delta^{seq}_{\forall\beta.\,(\beta\to\mathsf{Maybe}\,(\tau',\beta))\to\beta\to\tau,\emptyset}.$$

Expanding this statement according to Figure 4 yields that for every choice of closed types τ_1 and τ_2, an admissible, total, and left-closed relation $\mathcal{R} \in Rel^{seq}(\tau_1, \tau_2)$, functions $psi_1 :: \tau_1 \to \mathsf{Maybe}\ (\tau', \tau_1)$ and $psi_2 :: \tau_2 \to \mathsf{Maybe}\ (\tau', \tau_2)$, and values $e_1 :: \tau_1$

[6] The instantiation $g = (\lambda x\, y \to \textbf{case}\ x\ y\ \textbf{of}\ \mathsf{Just}\ z \to [])$, $psi = (\lambda x \to \textbf{case}\ x\ \textbf{of}\ [] \to \mathsf{Just}\ \bot)$, and $e = []$ demonstrates that (even in the absence of *seq* and even for strict *psi*) semantic equivalence does not hold in general. Svenningsson's paper does not mention this.

and $e_2 :: \tau_2$ the following holds:

$$\begin{array}{l}
\quad (g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot) \\
\wedge \, (\qquad (\mathit{psi}_1 \neq \bot \Rightarrow \mathit{psi}_2 \neq \bot) \\
\qquad\quad \wedge \, (\forall b_1 :: \tau_1, b_2 :: \tau_2. \\
\qquad\qquad (b_1, b_2) \in \mathcal{R} \Rightarrow (\mathit{psi}_1\ b_1, \mathit{psi}_2\ b_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',[\mathcal{R}/\beta]}, \mathcal{R}))) \\
\quad \Rightarrow \quad (g_{\tau_1}\ \mathit{psi}_1 \neq \bot \Rightarrow g_{\tau_2}\ \mathit{psi}_2 \neq \bot) \\
\qquad\quad \wedge \, ((e_1, e_2) \in \mathcal{R} \Rightarrow (g_{\tau_1}\ \mathit{psi}_1\ e_1, g_{\tau_2}\ \mathit{psi}_2\ e_2) \in \Delta^{seq}_{\tau,[\mathcal{R}/\beta]})).
\end{array}$$

Using the fact that for the closed types τ and τ' we have $\Delta^{seq}_{\tau,[\mathcal{R}/\beta]} = \Delta^{seq}_{\tau,\emptyset}$ and $\Delta^{seq}_{\tau',[\mathcal{R}/\beta]} = \Delta^{seq}_{\tau',\emptyset}$, dropping two conjuncts from the above, and strengthening one precondition, we obtain the following weaker statement:

$$\begin{array}{l}
\quad \mathit{psi}_2 \neq \bot \\
\wedge \, (\forall b_1 :: \tau_1, b_2 :: \tau_2. \\
\qquad (b_1, b_2) \in \mathcal{R} \\
\qquad \Rightarrow (\mathit{psi}_1\ b_1, \mathit{psi}_2\ b_2) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))) \\
\wedge \, (e_1, e_2) \in \mathcal{R} \\
\Rightarrow (g_{\tau_1}\ \mathit{psi}_1\ e_1, g_{\tau_2}\ \mathit{psi}_2\ e_2) \in \Delta^{seq}_{\tau,\emptyset}.
\end{array}$$

We consider two instantiations of this.

First, we instantiate

$$\begin{array}{lll}
\tau_1 = [\tau'], & \mathit{psi}_1 = \mathit{listpsi}, & e_1 = \mathit{unfoldr}\ \mathit{psi}\ e, \\
\mathcal{R} = \sqsubseteq ; (\mathit{unfoldr}\ \mathit{psi})^{-1}, & \mathit{psi}_2 = \mathit{psi}, & e_2 = e
\end{array}$$

for strict $\mathit{psi} :: \tau_2 \rightarrow \mathsf{Maybe}\ (\tau', \tau_2)$. Note that the instantiation for $\mathcal{R}$ is permissible because *unfoldr psi* is a strict function for strict *psi*. We obtain:

$$\begin{array}{l}
\quad \mathit{psi} \neq \bot \\
\wedge \, (\forall b_1 :: [\tau'], b_2 :: \tau_2. \\
\qquad b_1 \sqsubseteq \mathit{unfoldr}\ \mathit{psi}\ b_2 \\
\qquad \Rightarrow (\mathit{listpsi}\ b_1, \mathit{psi}\ b_2) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))) \\
\wedge \, \mathit{unfoldr}\ \mathit{psi}\ e \sqsubseteq \mathit{unfoldr}\ \mathit{psi}\ e \\
\Rightarrow (g_{[\tau']}\ \mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ e), g_{\tau_2}\ \mathit{psi}\ e) \in \Delta^{seq}_{\tau,\emptyset}.
\end{array}$$

Since *listpsi* is monotonic, so that $b_1 \sqsubseteq \mathit{unfoldr}\ \mathit{psi}\ b_2$ implies $\mathit{listpsi}\ b_1 \sqsubseteq \mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ b_2)$, and since $\sqsubseteq$ is transitive, we can prove that the second conjunct of the precondition holds by showing that

$$(\mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ b_2), \mathit{psi}\ b_2) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))$$

for every $b_2 :: \tau_2$. By the definitions of *unfoldr* and *listpsi* the element in the left position is equal to:

$$\begin{array}{l}
\textbf{case}\ \mathit{psi}\ b_2\ \textbf{of}\ \mathsf{Nothing} \quad \rightarrow \mathsf{Nothing} \\
\qquad\qquad\qquad \mathsf{Just}\ (a, b') \rightarrow \mathsf{Just}\ (a, \mathit{unfoldr}\ \mathit{psi}\ b').
\end{array}$$

By case distinction on the value of $\mathit{psi}\ b_2 :: \mathsf{Maybe}\ (\tau', \tau_2)$ we can check that this is indeed always related to $\mathit{psi}\ b_2$ by $\sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))$ as follows. The cases $\bot$ and $\mathsf{Nothing}$ are straightforward, using the reflexivity of $\sqsubseteq$ and the definition of $\mathit{lift}_{\mathsf{Maybe}}$. Similarly, the proof obligation in the case $\mathit{psi}\ b_2 = \mathsf{Just}\ (a, b')$ for some $a :: \tau'$ and $b' :: \tau_2$ reduces to:

$$((a, \mathit{unfoldr}\ \mathit{psi}\ b'), (a, b')) \in \sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}).$$

But this follows from reflexivity of $\sqsubseteq$, the definition of $\mathit{lift}_{(,)}$, law (7) for the closed type τ', and the instantiation of $\mathcal{R}$. Finally, in the case $\mathit{psi}\ b_2 = \mathsf{Just}\ \bot$,

$$(\bot, \mathsf{Just}\ \bot) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))$$

follows from $\bot \sqsubseteq \mathsf{Just}\ \bot$, the definition of $\mathit{lift}_{\mathsf{Maybe}}$, the reflexivity of $\sqsubseteq$, and the definition of $\mathit{lift}_{(,)}$. The third conjunct of the precondition in the above implication is trivially true, hence we obtain:

$$\mathit{psi} \neq \bot \Rightarrow (g_{[\tau']}\ \mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ e), g_{\tau_2}\ \mathit{psi}\ e) \in \Delta^{seq}_{\tau,\emptyset},$$

from which law (11) follows by the definition of *destroy*.

Second, we instantiate

$$\begin{array}{lll}
\tau_2 = [\tau'], & \mathit{psi}_1 = \mathit{psi}, & e_1 = e, \\
\mathcal{R} = (\mathit{unfoldr}\ \mathit{psi}) ; \sqsubseteq, & \mathit{psi}_2 = \mathit{listpsi}, & e_2 = \mathit{unfoldr}\ \mathit{psi}\ e
\end{array}$$

for strict and total $\mathit{psi} :: \tau_1 \rightarrow \mathsf{Maybe}\ (\tau', \tau_1)$ that never returns $\mathsf{Just}\ \bot$. Note that the instantiation for $\mathcal{R}$ is permissible because the conditions on *psi* guarantee that *unfoldr psi* is a strict and total function. We obtain:

$$\begin{array}{l}
\quad \mathit{listpsi} \neq \bot \\
\wedge \, (\forall b_1 :: \tau_1, b_2 :: [\tau']. \\
\qquad \mathit{unfoldr}\ \mathit{psi}\ b_1 \sqsubseteq b_2 \\
\qquad \Rightarrow (\mathit{psi}\ b_1, \mathit{listpsi}\ b_2) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))) \\
\wedge \, \mathit{unfoldr}\ \mathit{psi}\ e \sqsubseteq \mathit{unfoldr}\ \mathit{psi}\ e \\
\Rightarrow (g_{\tau_1}\ \mathit{psi}\ e, g_{[\tau']}\ \mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ e)) \in \Delta^{seq}_{\tau,\emptyset}.
\end{array}$$

The first and the third conjuncts of the precondition in this implication obviously hold. To establish the validity of the second conjunct, we note that for every $b_1 :: \tau_1$ and $b_2 :: [\tau']$, $\mathit{unfoldr}\ \mathit{psi}\ b_1 \sqsubseteq b_2$ and monotonicity of *listpsi* imply the following inequation:

$$\mathit{listpsi}\ (\mathit{unfoldr}\ \mathit{psi}\ b_1) \sqsubseteq \mathit{listpsi}\ b_2.$$

By the definitions of *unfoldr* and *listpsi* its left-hand side is equal to:

$$\begin{array}{l}
\textbf{case}\ \mathit{psi}\ b_1\ \textbf{of}\ \mathsf{Nothing} \quad \rightarrow \mathsf{Nothing} \\
\qquad\qquad\qquad \mathsf{Just}\ (a, b') \rightarrow \mathsf{Just}\ (a, \mathit{unfoldr}\ \mathit{psi}\ b').
\end{array}$$

Bearing in mind that neither *psi* nor *listpsi* ever returns $\mathsf{Just}\ \bot$ (the former by assumption, the latter by definition), it is easy to see from this that the inequation constrains the values of $\mathit{psi}\ b_1 :: \mathsf{Maybe}\ (\tau', \tau_1)$ and $\mathit{listpsi}\ b_2 :: \mathsf{Maybe}\ (\tau', [\tau'])$ to one of the following combinations:

$\mathit{psi}\ b_1$	$\mathit{listpsi}\ b_2$
$\bot$	$\bot$
$\bot$	$\mathsf{Nothing}$
$\mathsf{Nothing}$	$\mathsf{Nothing}$
$\bot$	$\mathsf{Just}\ (a', as')$
$\mathsf{Just}\ (a, b')$	$\mathsf{Just}\ (a', as') \mid a \sqsubseteq a' \wedge \mathit{unfoldr}\ \mathit{psi}\ b' \sqsubseteq as'$

It remains to be checked that in each of these cases the two values are related by $\sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq ; \mathit{lift}_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))$. This is an easy exercise using the reflexivity of $\sqsubseteq$, the facts that $\bot \sqsubseteq \mathsf{Nothing}$ and $\bot \sqsubseteq \mathsf{Just}\ (a', \bot)$ for every $a' :: \tau'$, the definitions of $\mathit{lift}_{\mathsf{Maybe}}$ and $\mathit{lift}_{(,)}$, law (7) for the closed type τ', the instantiation of $\mathcal{R}$, and strictness of *unfoldr psi*.

Having established the validity of all three conjuncts of the precondition in the above implication, its conclusion gives law (12) by the definition of *destroy*. □

The laws (11) and (12) are not yet the desired inequational versions of the *destroy*/*unfoldr* rule. This is because they depend on the relational interpretation of the closed type τ. In previous proofs of program transformations based on the fundamental property of a logical relation (e.g., in [6, 22]), such interpretations of closed types have silently been assumed to coincide with the relational interpretations of base types, i.e., with identity relations. This cannot be justified solely based on Wadler's parametricity theorem [23], from which these proofs claim to be derived, but rather requires Reynolds' *identity extension lemma* [17].

Reynolds also considered an "order-relation semantics" in which the interpretations of base types are semantic approximation relations, and noted that a corresponding extension lemma holds for it (prior to the inclusion of polymorphic types). This motivates the following conjecture:

if τ is a closed type, then:

$$\Delta^{seq}_{\tau,\emptyset} = \sqsubseteq_\tau \tag{13}$$

Coincidence of $\Delta^{seq}_{\tau,\emptyset}$ and $\sqsubseteq_\tau$ is easily established by induction for types τ not containing $\forall$-quantifications. This is because our logical relation interprets nonparametrized datatypes as $\sqsubseteq$, and because its relational actions for function types and parametrized datatypes preserve $\sqsubseteq$ (i.e., $\sqsubseteq_\tau \rightarrow^{seq} \sqsubseteq_{\tau'} = \sqsubseteq_{\tau\rightarrow\tau'}$ and, e.g., $\sqsubseteq_{\mathsf{Maybe}\,\tau} ; \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq_\tau) = \sqsubseteq_{\mathsf{Maybe}\,\tau}$).

To show that conjecture (13) also holds for types involving polymorphism is more complicated. Indeed, we encounter a problem analogous to that which arises in Section 8 of [17] for the identity extension lemma. To complete the induction step for $\forall$-types, one needs to assume the validity of a statement relating instances of a polymorphic value by the logical relation, interpreting the quantified type variable by an arbitrary (in our case: admissible, total, and left-closed) relation between the types at which instantiation occurs. Since in Section 2 we have not been explicit about which functions from types to values our semantic model contains at polymorphic types, this statement is not known to hold a priori. Reynolds solves the problem by incorporating precisely the required statement into the definition of the set of values a polymorphic type contains.[7] That no values of terms expressible in the underlying language are unduly excluded by doing so is then argued by appealing to the identity extension lemma itself, as well as to the abstraction theorem. Since the latter corresponds to the generalized form of Wadler's parametricity theorem in Section 6 of [23] and thus to the generalization of the fundamental property (7) for types and terms potentially containing free variables mentioned in our sketched proof in Section 6, the same approach is also expected to work in our setting.

Another approach would be to mirror Pitts' operational techniques [14]. He constructed a logical relation for a calculus very similar to the one without *seq* handled in Section 4 and proved that it interprets arbitrary closed types as contextual equivalence relations. This was used in [8, 9, 10] to give proofs of program transformations based on free theorems that make explicit the previously implicit use of the coincidence of relational interpretations of closed types with identity relations. In a message on the Types mailing list [2], Pitts suggested that changing the interpretations of base types to semantic approximations would give an analogue of (13). However, *seq* was not considered in that discussion.

Using the plausible assumption (13), the laws (11) and (12) turn into the following:

if $psi \neq \bot$ and psi is strict, then:

$$destroy\ g\ (unfoldr\ psi\ e) \sqsubseteq g\ psi\ e \tag{14}$$

if psi is strict and total and never returns $\mathsf{Just}\ \bot$*, then:*

$$destroy\ g\ (unfoldr\ psi\ e) \sqsupseteq g\ psi\ e \tag{15}$$

According to law (14), the *destroy*/*unfoldr* transformation is safe in the presence of *seq* if the first arguments of all occurrences of *unfoldr* in the original program are strict functions different from $\bot$. Most of the examples given in [20] satisfy this restriction, with the notable exceptions of a — rather toy — definition of the empty list as an *unfoldr* and the following function definition:

$$repeat\ x = unfoldr\ (\lambda a \rightarrow \mathsf{Just}\ (x, a))\ \bot$$

Fusion with this function as a producer can be problematic.

If *psi* is strict, total, distinct from $\bot$, and never returns $\mathsf{Just}\ \bot$, then laws (14) and (15) together guarantee that the *destroy*/*unfoldr* transformation is a semantic equivalence.

8.2 Short Cut Fusion

The classical program transformation proved with a free theorem is the *foldr*/*build* rule [6]. It states that for appropriately typed *g*, *c*, and *n*:

$$foldr\ c\ n\ (build\ g) = g\ c\ n \tag{16}$$

In the presence of *seq* this law fails, e.g., for the instantiation $g = seq$, $c = \bot$, and $n = []$. But under the assumption (13) we can prove that the following laws hold even when *seq* is present:

$$foldr\ c\ n\ (build\ g) \sqsupseteq g\ c\ n \tag{17}$$

if $c\ \bot\ \bot \neq \bot$ and $n \neq \bot$, then:

$$foldr\ c\ n\ (build\ g) = g\ c\ n \tag{18}$$

Law (17) gives only partial correctness of the *foldr*/*build* rule in general because the transformed program may be less defined than the original one. To recover total correctness in law (18), *c* and *n* must be restricted so that *foldr c n* is total (in addition to being strict). This coincides with what the conventional wisdom has to say about *foldr*/*build*.

8.3 The Concatenate Vanishes For Free

In [22] the function $vanish_{++}$ was given together with a proof of the following law for appropriately typed *g*, in the absence of *seq*:

$$g\ []\ (:)\ (++) = vanish_{++}\ g \tag{19}$$

Read from left to right, this law can be considered as a program transformation that eliminates concatenate operations from uniformly abstracted list producers.

In Appendix B of [22] it was noted that in the presence of *seq* the transformation might improve the termination behavior of programs. The sketched proof that the converse cannot happen anticipated some of the ideas from the present paper, but did not correctly handle **all** the subtleties that the presence of *seq* entails for proofs based on free theorems. With the logical relation constructed in Section 6, the fundamental property (7), and assumption (13), we now more rigorously obtain:

$$g\ []\ (:)\ (++) \sqsubseteq vanish_{++}\ g \tag{20}$$

Analogous inequational laws for other *vanish*-combinators given in [22] can also be proved in the presence of *seq*.

[7] In fact, Reynolds considers the added condition to draw the dividing line between parametric and ad-hoc polymorphism.

9 Directions for Future Research

In this paper we have investigated the impact that a polymorphic strict evaluation primitive, such as Haskell's *seq*, has on free theorems derivable from polymorphic types in a nonstrict functional language. The lessons learned may aid in determining the effects that the addition of other primitives, such as the ones used to incorporate I/O and stateful references in Haskell, has on free theorems.

To contain the weakening of free theorems due to *seq* so that it impacts only those functions that actually use *seq*, a qualified type system along the lines of [11] could be devised. The basic challenge here is to determine when to use the standard relational actions for interpreting function types or algebraic datatypes and when to use appropriately adapted relational actions for doing so.

While free theorems derived from our new logical relation also hold for programs which do not contain *seq* at all, they may be overly restrictive in such situations compared with free theorems obtained from the standard logical relation. On the other hand, using a (different) asymmetric logical relation could prove worthwhile also in that setting. For example, the strongest justification for the *destroy*/*unfoldr* rule in a nonstrict language without *seq* that could be proved in [8] was semantics-preservation for strict *psi* that never returns $\mathsf{Just}\ \perp$. But by employing the asymmetry idea it should also be possible to establish the inequational law (10) without preconditions. Similarly, it would be interesting to investigate what our approach has to contribute to the study of functional languages where strict rather than lazy evaluation is the default.

An alternative to the denotational approach taken in the current paper is Pitts' operational semantics-based approach to constructing parametric models of higher-order lambda calculi [14]. The delicate issue which arises in Pitts' approach to parametricity is tying the operational semantics of a calculus supporting new primitives into the relational interpretations of its types. The present paper can be seen as providing insight into the issues which are likely to arise when modifying the operational approach to accommodate *seq*, but the precise connections between the denotational style restrictions on relations reflected in our adapted logical relation and operational style closure operators as employed by Pitts remain topics for further investigation.

10 Acknowledgments

The ideas presented in this paper took shape during a visit of Patricia Johann to Dresden University of Technology, funded by the Graduiertenkolleg 334 of the "Deutsche Forschungsgemeinschaft".

11 References

[1] The Haskell Mailing List Archive
(`http://www.mail-archive.com/haskell@haskell.org`).

[2] The Types Forum List Archive
(`http://www.cis.upenn.edu/∼bcpierce/types/archives`), message `/current/msg00847.html`.

[3] O. Chitil. Type inference builds a short cut to deforestation. In *ICFP'99, Proc.*, pages 249–260. ACM Press.

[4] S. Cosmadakis, A. Meyer, and J. Riecke. Completeness for typed lazy inequalities. In *LICS'90, Proc.*, pages 312–320. IEEE Computer Society Press.

[5] H. Friedman. Equality between functionals. In *Logic Colloquium '72–73, Proc.*, pages 22–37. Springer-Verlag.

[6] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA'93, Proc.*, pages 223–232. ACM Press.

[7] J.-Y. Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

[8] P. Johann. On proving the correctness of program transformations based on free theorems for higher-order polymorphic calculi. To appear in *Math. Struct. in Comp. Sci.*

[9] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symb. Comp.*, 15:273–300, 2002.

[10] P. Johann. Short cut fusion is correct. *J. Funct. Prog.*, 13:797–814, 2003.

[11] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *ESOP'96, Proc.*, pages 204–218. Springer-Verlag.

[12] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[13] S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: A common intermediate language for ML and Haskell. In *POPL'98, Proc.*, pages 49–61. ACM Press.

[14] A. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. in Comp. Sci.*, 10:321–359, 2000.

[15] G. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.

[16] J. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation '74, Proc.*, pages 408–423. Springer-Verlag.

[17] J. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83, Proc.*, pages 513–523. Elsevier Science Publishers B.V.

[18] D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[19] R. Statman. Logical relations and the typed lambda-calculus. *Inf. and Control*, 65:85–97, 1985.

[20] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP'02, Proc.*, pages 124–132. ACM Press.

[21] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Prog.*, 8:23–60, 1998.

[22] J. Voigtländer. Concatenate, reverse and map vanish for free. In *ICFP'02, Proc.*, pages 14–25. ACM Press.

[23] P. Wadler. Theorems for free! In *FPCA'89, Proc.*, pages 347–359. ACM Press.

[24] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL'89, Proc.*, pages 60–76. ACM Press.