Authors' version of a paper appearing in *Fundamenta Informaticae 69 (1–2), pp. 63–102, IOS Press, 2006*

The Impact of seq on Free Theorems-Based Program Transformations*

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, standard parametricity results — including so-called free theorems — fail for nonstrict languages supporting a polymorphic strict evaluation primitive such as Haskell's seq. A folk theorem maintains that such results hold for a subset of Haskell corresponding to a Girard-Reynolds calculus with fixpoints and algebraic datatypes *even when* seq *is present* provided the relations which appear in their derivations are required to be bottom-reflecting and admissible. In this paper we show that this folklore is incorrect, but that parametricity results *can* be recovered in the presence of seq by restricting attention to left-closed, total, and admissible relations instead. The key novelty of our approach is the asymmetry introduced by left-closedness, which leads to "inequational" versions of standard parametricity results together with preconditions guaranteeing their validity even when seq is present. We use these results to derive criteria ensuring that both equational and inequational versions of short cut fusion and related program transformations based on free theorems hold in the presence of seq.

*This work is based on the paper Free Theorems in the Presence of seq, which appeared in *31st Symposium on Principles of Programming Languages (POPL'04), Proceedings*, © ACM Press, 2004. http://doi.acm.org/10.1145/964001.964010
†Research supported in part by the National Science Foundation under grant CCF-0429072.
‡Research supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-4.

Keywords: Haskell, control primitives, correctness proofs, denotational semantics, functional programming languages, logical relations, mixing strict and nonstrict evaluation, parametricity, polymorphism, program transformations, rank-2 types, short cut fusion, theorems for free

1. Introduction

Program transformation is a key technology in software engineering. It has applications in program synthesis, compilation, optimization, refactoring, software renovation, and reverse engineering, and figures significantly in current trends such as intentional programming, aspect-oriented programming, and, more generally, generative programming [8]. Automatic program transformation enables programmers to work at a higher level of abstraction than would otherwise be possible, thus giving rise to increases in programmer productivity. It can also lead to increases in reliability, maintainability, reusability, and efficiency of software. For a survey of the broad area of program transformation and state-of-the-art contributions describing its impact on (applied) software engineering see, e.g., [1] and [4].

Program transformation is, however, only useful if there is a clear relationship between the computational meanings of programs to be transformed and the programs obtained by transforming them. Programmers need to be able to reason about their programs with confidence that automatically transforming them will not change their semantics in unexpected ways. One approach to formalizing the kind of semantic statements which ensure that this is possible is based on *free theorems*.

Free theorems-based transformations

Ever since they were first popularized by Wadler [36], free theorems have been used to derive program equivalences involving parametrically polymorphic functions in (nonstrict functional) programming languages based on the (nonstrict) Girard-Reynolds lambda calculus $\lambda^{\forall}$ [11, 27]. A free theorem is considered *free* because it can be derived solely from the type of a function, with no knowledge whatsoever of the function's actual definition. In essence, a free theorem records a constraint arising from the fact that a parametrically 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.

Particularly effective use of free theorems has been made for justifying program transformations based on *rank-2 polymorphic* [18] combinators, i.e., on functions taking polymorphic functions as arguments. Perhaps the best-known example of such a transformation is *short cut fusion* [10]. It uses a single local replacement rule to eliminate intermediate lists between producers and consumers of special forms, and thus to mitigate the tension between modularity and efficiency. Other performance-improving transformations are based on similar rules for fusing consumers of algebraic data structures other than lists with producers parametrized over substitution values [12], rules which are category-theoretic duals of these [32, 33], and rules for eliminating data-manipulating operations other than data constructors [35].

Free theorems can fail in the presence of strict evaluation

Free theorems hold unconditionally for functions in $\lambda^{\forall}$. But for calculi that more closely resemble modern functional languages the story is not so simple. It is well known [17, 36], for example, that adding a fixpoint primitive to a calculus weakens its free theorems by imposing admissibility — i.e., strictness and continuity — conditions on (some of the) functions appearing in them. Algebraic datatypes and

pattern matching also impose strictness constraints. Moreover, to help programmers control the time and space behavior of programs, nonstrict languages often provide primitives for introducing strict evaluation into computations. These can compromise free theorems even further.

For example, Haskell is a nonstrict language, so that function arguments are evaluated only when required. But evaluation can be explicitly forced using the polymorphic strict evaluation primitive seq, which satisfies the following specification:

$$\begin{aligned} &seq \perp b = \perp \\ &seq\ a\ \ b = b\text{ , if } a \neq \perp \end{aligned}$$

Here $\perp$ 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. A simple example using seq to improve performance is the following accumulating sum function:

$$\begin{aligned} &accsum :: [\mathsf{Int}] \to \mathsf{Int} \to \mathsf{Int} \\ &accsum\ []\qquad\ \ y = y \\ &accsum\ (x : xs)\ y = seq\ acc\ (accsum\ xs\ acc)\ \mathbf{where}\ acc = x + y \end{aligned}$$

where seq makes sure that the accumulating parameter is computed immediately in every recursive step, rather than unnecessarily building up and maintaining a complex closure representing the overall sum, which would then be computed only in the end. See [34] for further examples of programs which use seq. Other means of explicitly introducing strictness in Haskell programs — e.g., strict datatypes and the strict application function $! — are all definable in terms of seq.

Haskell 98 gives seq the type `a -> b -> b` (the Haskell equivalent of $\forall\alpha\ \beta.\ \alpha \to \beta \to \beta$), suggesting that it is a parametrically polymorphic function. By contrast, earlier versions of Haskell give seq the qualified type `Eval a => a -> b -> b`, which reflects the fact that seq is indeed polymorphic, although not necessarily parametrically so. Although the earlier typing is more accurate than the Haskell 98 typing, the class context was dropped from Haskell 98 because it was ultimately considered too burdensome for the programmer: seq is typically used to tune the performance of programs *after* they are written, and it was argued that this should not affect their types. The Haskell 98 report [21] is, however, careful to warn that the provision of fully polymorphic seq has important semantic consequences. The following example demonstrates that strict evaluation can break even simple free theorems in dramatic and unexpected ways. The effect on the more complex free theorems underlying short cut fusion and related transformations discussed in Section 8 is similar.

A free theorem given in Figure 1 of [36] states that for *any* function

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

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

$$filter\ p\ (map\ h\ l) = map\ h\ (filter\ (p \circ h)\ l) \qquad (1)$$

As usual, $map\ h$ applies the function h to every element of a list, and $\circ$ is function composition. For the definition of $filter$ from Haskell's standard prelude, which specifies that $filter\ p$ filters in all elements of

a list which satisfy the predicate p, law (1) thus formalizes the observation that mapping h over a list and then filtering the resulting list by p gives the same result as mapping h over the list obtained by filtering the original one by $p \circ h$. Haskell definitions of the functions used in law (1), as well as those of other functions and types used in this paper, appear in Figure 1 below.

$$
\begin{array}{l}
\mathbf{data}\ \mathsf{Bool} \quad = \mathsf{False} \mid \mathsf{True} \\
\mathbf{data}\ \mathsf{Maybe}\ \alpha = \mathsf{Nothing} \mid \mathsf{Just}\ \alpha \\
\\
\mathit{map} :: \forall \alpha\ \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\mathit{map}\ h = f\ \mathbf{where}\ f\ [] \quad = [] \\
\qquad\qquad\qquad\qquad f\ (x : xs) = h\ x : f\ xs \\
\\
\mathit{filter} :: \forall \alpha.\ (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
\mathit{filter}\ p = f\ \mathbf{where}\ f\ [] \quad = [] \\
\qquad\qquad\qquad\qquad f\ (x : xs) = \mathbf{case}\ p\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{True} \ \to x : f\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{False} \to f\ xs \\
\\
(\circ) :: \forall \alpha\ \beta\ \gamma.\ (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\
f_1 \circ f_2 = (\lambda x \to f_1\ (f_2\ x)) \\
\\
\mathit{fix} :: \forall \alpha.\ (\alpha \to \alpha) \to \alpha \\
\mathit{fix}\ g = g\ (\mathit{fix}\ g) \\
\\
\mathit{id} :: \forall \alpha.\ \alpha \to \alpha \\
\mathit{id}\ x = x \\
\\
(+\!\!+) :: \forall \alpha.\ [\alpha] \to [\alpha] \to [\alpha] \\
[] \quad +\!\!+ ys = ys \\
(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{array}
$$

Figure 1. Some Haskell definitions.

Because they embody properties that *all* terms of a given polymorphic type must satisfy, free theorems are quite general. But this generality is also their potential downfall in the presence of seq. If, for example, we implement a "sequentialized" function with *filter*'s type in the fragment of Haskell corresponding to $\lambda^{\forall}$ with (nonstrict) algebraic datatypes, a fixpoint operator *fix* (defined as in Figure 1), and

seq by

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

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

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

In each case, law (1) fails because one of its two sides is less defined than the other. The direction in which the equation becomes a strict inequation with respect to the semantic approximation order $\sqsubseteq$ is indicated for each instantiation. A file containing all instantiations used as counterexamples in this paper, as well as short scripts of their execution in the Haskell interpreter Hugs, is available online [2].

Although one might suspect that the use of fixpoint recursion or pattern matching is responsible for the breakdown of law (1) for the sequentialized filter function above, careful analysis shows that it is the use of seq which is actually to blame. Launchbury and Paterson [17] 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 admissibility conditions are required in free theorems for functions involving fixpoints and pattern matching. For example, the **case**-expressions in the sequentialized definition of filter — which produce $\bot$ on selectors that are themselves undefined — have the return type $[\alpha]$, which the type system of Launchbury and Paterson recognizes is pointed without any condition on α. Similarly, the use of fix rather than a directly recursive definition makes it clear that the recursion takes place at type $[\alpha] \to [\alpha]$, which Launchbury and Paterson's type system again recognizes is pointed without any condition on α since $[\alpha]$ is. These observations can be used to show that law (1) holds without any conditions on p, h, or l, provided all invocations of seq are dropped — in which case we get precisely the filter function from Figure 1. (By contrast, if 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. A similar remark holds for uses of pattern matching with return types whose pointedness relied on pointedness of α.) This shows that it is not the use of recursion or pattern matching in our sequentialized definition of filter that is responsible for the breakdown of law (1). The evil really does reside in seq.

It is also worth noting that different ways of using seq are responsible for the failure of law (1) for the four instantiations given above. For the first instantiation, 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 $p \circ h = \bot \circ id = (\lambda x \to \bot) \neq \bot$. For the second instantiation, 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 around for the third instantiation. Finally, for 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 although our sequentialized definition of $filter$ does not use recursion or pattern matching in a way that requires α to be pointed, the associated strictness condition on h creeps in through the back door via seq here.

Recovering free theorems by asymmetry

The failure of free theorems in the presence of seq has been noted before (see, e.g., Section 6.2 of [21], Section 5.3 of [23], Appendix B of [35], and discussions on the Haskell mailing list [3]), but the extent to which it compromises Haskell's parametricity properties has not been studied thoroughly. A folk theorem — expressed, e.g., in [23] — has for nearly a decade had it that a free theorem remains valid in the presence of seq if all of the functions appearing in it (where one is free to make a choice) are strict and total. This implies, for example, that law (1) should hold for every strict and total h. But the first instantiation of (1) above shows that this claim does not hold.

This paper does more, however, than just demonstrate that the folkloric approach to free theorems in Haskell with seq is incorrect. It also provides valid criteria for determining when free theorems hold in the presence of seq and applies them to study the impact of seq on the correctness of program transformations based on these theorems. To recover free theorems when seq is present, an asymmetry is introduced into the standard symmetric approach to deriving them. This allows the derivation, for each standard equational free theorem which holds in the absence of seq, of two related "inequational" free theorems that hold even when seq is present. For example, this asymmetric approach is used in Section 7 to prove that, for every function of $filter$'s type and all appropriately typed p, h, and l, the following inequational versions of law (1) hold even in the presence of seq.

Theorem 1.1. For every function

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

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

if h is strict, then:

$$filter\ p\ (map\ h\ l) \sqsubseteq map\ h\ (filter\ (p \circ h)\ l) \tag{2}$$

if $p \neq \bot$ and h is strict and total, then:

$$filter\ p\ (map\ h\ l) \sqsupseteq map\ h\ (filter\ (p \circ h)\ l) \tag{3}$$

If $p \neq \bot$ and h is strict and total, then laws (2) and (3) together guarantee that law (1) holds even when seq is present. That is, even in the presence of seq:

if $p \neq \bot$ *and* h *is strict and total, then:*

$$filter\ p\ (map\ h\ l) = map\ h\ (filter\ (p \circ h)\ l) \tag{4}$$

Inequational free theorems like those for $filter$ can be derived for other functions as well. Such theorems can be used to prove, for example, that the (unrestricted) $foldr/build$ rule used in short cut fusion is partially correct in the presence of seq, which is a genuinely new result. They can also be used to derive conditions under which the converse semantic inequality — and, therefore, total correctness of the $foldr/build$ rule — is guaranteed. (See Section 8.2 for details.) While partial correctness results may be primarily of theoretical interest, the other direction in which the equational free theorem underlying a program transformation can turn into an inequation is interesting from both theoretical and practical points of view. Indeed, it is customary to require program transformations to be semantics-preserving, but in applications it is often enough to know that programs obtained by applying transformations are semantically approximated by those from which they are derived, in the sense that the former are always at least as defined as the latter. The use of transformations potentially improving the termination behavior of programs in an optimizing compiler could easily be controlled via a command line switch. They would not necessarily be employed during program development and testing so as not to hide algorithmic mistakes from detection, but could nevertheless fire during final "production runs".

The move from the equational to the inequational setting might appear at first glance to be an undesirable weakening of parametricity results, but it actually adds important strength: we obtain *more* free theorems than would be available through a less radical revision of the folkloric approach, and inequational free theorems — like (2) and (3) — can often be combined to provide criteria — such as the preconditions of (4) — under which even their equational versions hold in the presence of seq. That there is no way to discard the inequational approach altogether without losing information is witnessed by the analysis of the law for $vanish_{++}$ from [35] carried out in Section 8.3. That analysis shows that it is impossible to preserve an equational form of the law for $vanish_{++}$ in the presence of seq while also preserving its nature as a free theorem, i.e., while maintaining its dependence on just the polymorphic type of the argument to $vanish_{++}$. Thus, while the asymmetric approach can still derive a useful inequational law, any approach insisting that all laws be equational would be doomed to complete failure.

This paper

This paper differs substantially from the conference version on which it is based [15]. Several proofs which were omitted from the conference version due to space limitations have been included, and laws (13) and (16) have been generalized slightly. Moreover, the specific application of free theorems to program transformations is now discussed in detail. Examples are used to show that none of the additional preconditions generated in derivations of results that hold in the presence of seq can be dropped, and the severity of these preconditions from a practical, program transformation-oriented, point of view is considered. Finally, the potential impact of our results on the largely unexplored topic of free theorems in purely strict languages is discussed.

The remainder of this paper is organized as follows. Section 2 briefly considers the functional language we use, and Section 3 introduces auxiliary notions and definitions. Section 4 recalls how free theorems are obtained in the absence of seq, while Sections 5 and 6 motivate and develop our approach to parametricity — and thus free theorems — in its presence. Section 7 shows how this approach can

be used to derive the inequational free theorems (2) and (3). The focus of the paper is Section 8, which considers in depth how short cut fusion and related free theorems-based program transformations can be compromised by seq, and shows how our approach to free theorems provides a formal basis for such transformations when seq is present. The results obtained are evaluated from a pragmatic point of view. Section 9 concludes and proposes directions for future research.

2. Functional language

We use a subset of the pure nonstrict functional programming language Haskell [21] that corresponds to $\lambda^{\forall}$ 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 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 parametrically polymorphic term t to a *closed type* (one without free variables) τ is made explicit, it is denoted by t_τ. The kind of *ad hoc* polymorphism provided by Haskell type classes [37] is not considered here, but Section 3.4 of [36] briefly discusses how it can be taken into account when deriving free theorems.

This paper does not aim to provide a model for full Haskell. This goal would be out of reach because there does not yet exist a formal semantics for Haskell, independent of any concrete implementation, against which to validate a model. Nevertheless, it is common practice to use a denotational style [29] for reasoning about Haskell programs. Indeed, although it is rarely explicitly acknowledged, most papers containing results about Haskell programs are based on a "naive" model such as might be obtained by extending the denotational semantics of [36]. In investigations of free theorems-based transformations for Haskell, it is further assumed — without proof and, again, typically without explicit mention — that this naive model is parametric. This paper follows in the same tradition: it investigates, under the widely held assumption that a parametric extension of the model in [36] exists for Haskell without seq, exactly how much of the power of free theorems can be retained when seq is thrown in. The assumption that a parametric model for Haskell exists is not unreasonable. It is justified in part by Pitts' recent operational semantics-based construction of a parametric model for PolyFix, a nonstrict polymorphic lambda calculus with fixpoints and "lazy" algebraic datatypes [25]. The subset of Haskell to which we restrict attention in this paper corresponds to PolyFix with seq; formalizing the intuitions developed here by extending Pitts' construction for PolyFix to incorporate seq is ongoing work.

The naive semantics for Haskell requires the semantic approximation relation $\sqsubseteq$ — interpreted as "less than or equally as defined as" — between values of the same type, and the value $\bot$ — 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 $\bot$, and limits of all chains. As in Haskell, algebraic datatypes and function spaces are lifted; in particular, λ-abstractions — and, therefore, partial applications of functions — are always distinct from $\bot$. 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, so that 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 $\sqsubseteq_\tau, \sqsubset_\tau \in Rel(\tau, \tau)$ and the value $\bot_\tau :: \tau$ 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 $\bot :: \tau$ at a particular closed type should not be confused with the polymorphic value $\bot :: \forall\alpha.\, \alpha$.

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:

$$(\forall\tau.\ u_\tau = \bot_{\tau_1[\tau/\alpha]}) \Rightarrow u = \bot_{\forall\alpha.\,\tau_1} \tag{5}$$

This observation will be used in Section 6 below.

To reduce the need for parentheses when writing down logical formulas, we use the conventions that the bodies of $\forall$- and $\exists$-quantifications extend as far as possible to the right and that conjunction ($\wedge$) takes precedence over implication ($\Rightarrow$).

A relation is *strict* if it contains the pair $(\bot, \bot)$. A relation is *total* if, for every pair (x, y) contained in it, $y = \bot$ implies $x = \bot$. A relation is *bottom-reflecting* if, for every pair (x, y) contained in it, $y = \bot$ iff $x = \bot$. 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 $h\ x \sqsubseteq h\ y$. These properties of relations are used in deriving our new parametricity results.

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 $\sqsubset^{-1}$ by $\sqsupseteq$ and $\sqsupset$, 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 (cf. Appendix A.1).

The lifting of $\mathcal{R} \in Rel(\tau_1, \tau_2)$ 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 (cf. Appendix A.2).

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:

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

We take $\mathit{lift}_{[]}(\mathcal{R})$ to be the largest, rather than the smallest, such relation in order to account for infinite lists. According to this definition, two lists are related by $\mathit{lift}_{[]}(\mathcal{R})$ 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 $\mathit{lift}_{[]}(\mathcal{R})$. If $\mathcal{R}$ is left-closed in addition, then $\sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R})$ is also continuous (cf. Appendix A.3). Note that in the special case that $\mathcal{R}$ is the graph of a Haskell function h, the relation $\mathit{lift}_{[]}(\mathcal{R})$ coincides with the graph of the function $map\ h$ as defined in Figure 1.

The following results are immediate from the definitions of the preceding paragraphs. Since $\sqsubseteq$ is reflexive, for all appropriately typed functions h and lists l:

$$(map\ h\ l, l) \in \sqsubseteq ; \mathit{lift}_{[]}(\sqsubseteq ; h^{-1}) \tag{6}$$
$$(l, map\ h\ l) \in \sqsubseteq ; \mathit{lift}_{[]}(h ; \sqsubseteq) \tag{7}$$

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 $\mathit{lift}_{[]}(\sqsubseteq ; h^{-1})$ in $\sqsubseteq ; (map\ h)^{-1}$ that:

$$(l_1, l_2) \in \sqsubseteq ; \mathit{lift}_{[]}(\sqsubseteq ; h^{-1}) \Rightarrow l_1 \sqsubseteq map\ h\ l_2 \tag{8}$$

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

$$(l_2, l_1) \in \sqsubseteq ; \mathit{lift}_{[]}(h ; \sqsubseteq) \Rightarrow map\ h\ l_2 \sqsubseteq l_1 \tag{9}$$

These observations are used to derive the inequational free theorems (2) and (3) in Section 7.

4. Free theorems in the absence of seq

The key to deriving free theorems from types is to interpret types as relations (as opposed to sets). Each type variable is thus interpreted as a relation, and associated with every n-ary type constructor is a map which produces a new relational interpretation from n given ones. Such a map is called a *relational action*. It is standard to interpret the base types in a language as identity relations, and to obtain the interpretations for non-base types via relational actions which propagate relations up the type hierarchy in a straightforward "extensional" manner. Using relational actions to propagate the relations interpreting type variables and constants up the type hierarchy builds a (type-indexed) *logical relation* [9, 26, 31], for which a parametricity theorem can be proved. 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.

The standard logical relation for $\lambda^{\forall}$ forms the theoretical basis of the usual equational free theorems. Its relational actions for its type constructors are given as follows. The relational action corresponding to the function type constructor maps 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.

To specify the standard relational action for the $\forall$-type constructor, 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 maps $\mathcal{F}$ to the following relation in $Rel(\forall\alpha.\,\tau_1, \forall\alpha.\,\tau_2)$:

$$\forall\mathcal{R} \in Rel.\, \mathcal{F}(\mathcal{R}) = \{(u, v) \mid \forall\tau_1', \tau_2', \mathcal{R} \in Rel(\tau_1', \tau_2').\, (u_{\tau_1'}, v_{\tau_2'}) \in \mathcal{F}(\mathcal{R})\}.$$

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 [28, 36], from which the standard free theorems are derived, then states that:

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

$$(t, t) \in \Delta_{\tau,\emptyset} \tag{10}$$

This is also called the *fundamental property* of the logical relation. For each closed term t of closed type τ we call the assertion that $(t, t) \in \Delta_{\tau,\emptyset}$ the *parametricity property* for t.

Proofs of parametricity theorems proceed by induction on the syntactic structure of terms, driven by type assignment rules, with the constants occurring in a language forming the base cases. 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.

4.1. Adding fixpoints and datatypes

Because fixpoints are interpreted as limits of chains based on $\bot$, 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.

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

Figure 3. Standard relational interpretations for datatypes.

For each algebraic datatype we need 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; other uses of pattern matching in Haskell programs — e.g., on left-hand sides of function equations — can all be translated into **case**-expressions. Proving that the fundamental property of the logical relation remains intact amounts to 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 [17].

Like the relational actions for the function and $\forall$-type constructors, the relational actions for alge-

braic 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 added to the language as well.

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. This parametricity property is obtained as follows:

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

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

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

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

Similar situations arise for each instantiation of law (1) given in Section 1. In the first instantiation, for example, we have that $p = \bot :: \tau \to \mathsf{Bool}$ and $h = id :: \tau \to \tau$ for some closed type τ. Since $\bot\ (id\ x) = \bot\ x$ for every $x :: \tau$, the sequentialized *filter* function's local definition of $fix\ g$ for $filter\ (p \circ h)$ is the same as the one for $filter\ p$. Thus, if

$$
\begin{aligned}
\mathcal{R} = \Delta_{\alpha\to\mathsf{Bool},[h/\alpha]} &= \{(p_1,p_2) \mid \forall x :: \tau.\ p_1\ x = p_2\ x\} \in Rel(\tau\to\mathsf{Bool},\tau\to\mathsf{Bool}) \\
\mathcal{S} = \Delta_{[\alpha]\to[\alpha],[h/\alpha]} &= \{(f_1,f_2) \mid \forall l :: [\tau].\ f_1\ l = f_2\ l\} \in Rel([\tau]\to[\tau],[\tau]\to[\tau])\,,
\end{aligned}
$$

then $\mathcal{R}$ and $\mathcal{S}$ are both admissible relations and $(p \circ h, p) \in \mathcal{R}$ and $(fix\ g, fix\ g) \in \mathcal{S}$. Nevertheless, $((p \circ h)\ `seq`\ (fix\ g), p\ `seq`\ (fix\ g)) \in \mathcal{S}$ — i.e., $(filter\ (p \circ h), filter\ p) \in \mathcal{S}$ — does not hold. The crucial observation is that $\mathcal{R}$ is *not* the identity relation on type $\tau \to \mathsf{Bool}$. Indeed, it contains the pair $((\lambda x \to \bot), \bot)$, whose two components are different in the presence of seq.

Since seq violates the parametricity property dictated by its type, other terms that are built using seq might do so as well. In fact, this is precisely what causes law (1) to fail for the sequentialized

$filter$ function from the introduction. The problem lies with relations, such as the instantiations of $\mathcal{R}$ in this section, that relate $\bot$ and non-$\bot$ values. It has therefore been proposed (e.g., in [17]) that quantified relations should be restricted from admissible ones to bottom-reflecting admissible ones. But the first counterexample in the introduction shows that this is not 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 fails 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, note 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 account for the difference between an 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 [7]. But rather than requiring bottom-reflectingness, we add totality and left-closedness to the admissibility restriction on relational interpretations. Thus, instead of just recovering the usual equational free theorems under quite severe preconditions, we are able to derive inequational versions of them under weaker ones.

6. Recovering free theorems in the presence of seq

As demonstrated in the previous section, the presence of seq is problematic if relational interpretations of types are allowed to relate $\bot$ and non-$\bot$ values. One way to accommodate 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, seq has the potential to make one of the two sides of a free theorem provable in its absence less defined than it would otherwise be (indeed potentially $\bot$), even while the other side remains unchanged. To derive interesting statements for these situations, we should therefore be more liberal.

To this end, we modify the standard logical relation in such a way that one of the two argument positions of a relational interpretation is "favored" with respect to definedness. An asymmetry is introduced 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 [35]. However, like

bottom-reflectingness, those restrictions are not preserved by all relational actions. Moreover, their *ad hoc* nature makes it uncertain whether they would provide a good starting point for putting things right by adjusting the relational actions.

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, we turn our attention to relations which are left-closed in addition to being admissible and total. More specifically, we interpret base types as semantic approximation relations rather than as identity relations, noting that semantic approximation relations satisfy all three of the imposed restrictions. In addition, 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 all three restrictions. Of course, 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 for the new logical relation, and must therefore have an "extensional flavor" which ties them to the semantics of the language. (In the absence of this consideration, every relational action could simply return the trivial relation $\{(\bot, \bot)\}$, but this clearly is not what we want.) 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 $Rel^{seq}(\tau_1 \to \tau_1', \tau_2 \to \tau_2')$:

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

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 corresponding to the $\forall$-type constructor maps $\mathcal{F}$ to the following relation in $Rel^{seq}(\forall\alpha.\, \tau_1, \forall\alpha.\, \tau_2)$:

$$\forall \mathcal{R} \in Rel^{seq}.\, \mathcal{F}(\mathcal{R}) = \{(u, v) \mid \forall \tau_1', \tau_2', \mathcal{R} \in Rel^{seq}(\tau_1', \tau_2').\, (u_{\tau_1'}, v_{\tau_2'}) \in \mathcal{F}(\mathcal{R})\}.$$

That the resulting relation is admissible follows from monotonicity and continuity of type instantiation in Haskell and from admissibility of all the $\mathcal{F}(\mathcal{R})$. Totality 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 $\sqsubseteq_\tau \in Rel^{seq}(\tau,\tau)$ for every closed type τ, we have that, for each such τ, $(u_\tau, (\bot_{\forall\alpha.\,\tau_2})_\tau) \in \mathcal{F}(\sqsubseteq_\tau)$, i.e., $(u_\tau, \bot_{\tau_2[\tau/\alpha]}) \in \mathcal{F}(\sqsubseteq_\tau)$. By totality of $\mathcal{F}(\sqsubseteq_\tau)$ we have $u_\tau = \bot_{\tau_1[\tau/\alpha]}$ for every such τ, and so by law (5) 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 observations 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 $\sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$, $\sqsubseteq ; \mathit{lift}_{(,)}(\mathcal{R}, \mathcal{S})$, and $\sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R})$ are also continuous (cf. Appendix A).

Hence, all relations that turn up in the definition of the new logical relation 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.

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

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

We now verify that our modified logical relation still has the following fundamental property, from which the new free theorems will be derived.

Theorem 6.1. Let Δ^{seq} be defined as in Figure 4. Then:

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

$$(t, t) \in \Delta^{seq}_{\tau,\emptyset} \tag{11}$$

Proof Sketch:
We first check that this law holds for seq. The parametricity property of seq according to the new logical relation is derived as follows:

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

The two implications $seq_{\tau_1\,\tau_1'}\neq\bot\Rightarrow seq_{\tau_2\,\tau_2'}\neq\bot$ and $seq_{\tau_1\,\tau_1'}\,a_1\neq\bot\Rightarrow seq_{\tau_2\,\tau_2'}\,a_2\neq\bot$ 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 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 trivially 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 $(xs,ys)\in\sqsubseteq;\mathit{lift}_{[]}(\mathcal{R})$ that $(x:xs, y:ys)\in\sqsubseteq;\mathit{lift}_{[]}(\mathcal{R})$. The former is obvious; to prove the latter is an easy exercise using the monotonicity of $(:)$. Similar arguments hold for the other data constructors.

What remains to be done is to mirror Wadler's sketched proof [36] that the term-forming operations of the polymorphic lambda calculus — λ-abstraction, function application, type abstraction, and 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. Thus, in comparison with 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 enough to prove the induction conclusion. For the other induction step involving a $\forall$-type — i.e., the one corresponding to type abstraction — no additional argument is needed.

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 $(\lambda\, x \to [\![t]\!]_{\varrho_1}) \neq \bot$ implies $(\lambda\, x \to [\![t]\!]_{\varrho_2}) \neq \bot$ 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 $[\![t]\!]_{\varrho_1}$ and $[\![t]\!]_{\varrho_2}$ 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 distinct from $\bot$.

The induction step for $\mathbf{case}$-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 $\mathbf{case}$-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 4.1. Since each new interpretation of an algebraic datatype is the composition of the semantic approximation ordering and its standard structural interpretation, the argument also 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 [36] ensure that $\bot$ and fix also fulfill their parametricity properties with respect to the new logical relation. Putting everything together, we conclude that (11) holds in the presence of seq, algebraic datatypes, and general fixpoint recursion.

6.1. Manufacturing permissible relations

A strategy often employed 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 (and, of course, the totality precondition where required) 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 the two inequational free theorems (2) and (3) in Theorem 1.1 even when seq is present.

Proof of Theorem 1.1:
The parametricity property derived from $filter$'s type is the following instance of law (11):

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

Expanding this statement according to 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 $\mathcal{R} \in Rel^{seq}(\tau_1, \tau_2)$, 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:

$$\begin{aligned}
&(filter_{\tau_1} \neq \bot \Rightarrow filter_{\tau_2} \neq \bot)\\
\wedge\ &((p_1 \neq \bot \Rightarrow p_2 \neq \bot) \wedge (\forall(x_1, x_2) \in \mathcal{R}.\ p_1\ x_1 \sqsubseteq p_2\ x_2)\\
&\Rightarrow\quad (filter_{\tau_1}\ p_1 \neq \bot \Rightarrow filter_{\tau_2}\ p_2 \neq \bot)\\
&\quad\wedge ((l_1, l_2) \in \sqsubseteq ; lift_{[]}(\mathcal{R}) \Rightarrow (filter_{\tau_1}\ p_1\ l_1,\ filter_{\tau_2}\ p_2\ l_2) \in \sqsubseteq ; lift_{[]}(\mathcal{R}))).
\end{aligned}$$

Using properties of $\wedge$ and $\Rightarrow$ to drop the two conjuncts $filter_{\tau_1} \neq \bot \Rightarrow filter_{\tau_2} \neq \bot$ and $filter_{\tau_1}\ p_1 \neq \bot \Rightarrow filter_{\tau_2}\ p_2 \neq \bot$ from the above, and also replacing the precondition $p_1 \neq \bot \Rightarrow p_2 \neq \bot$ with the stronger precondition $p_2 \neq \bot$, we obtain the following weaker statement:

$$\begin{aligned}
&p_2 \neq \bot \wedge (\forall(x_1, x_2) \in \mathcal{R}.\ p_1\ x_1 \sqsubseteq p_2\ x_2)\\
&\Rightarrow ((l_1, l_2) \in \sqsubseteq ; lift_{[]}(\mathcal{R}) \Rightarrow (filter_{\tau_1}\ p_1\ l_1,\ filter_{\tau_2}\ p_2\ l_2) \in \sqsubseteq ; lift_{[]}(\mathcal{R})).
\end{aligned}$$

We consider two instantiations of this, guided by the structure of laws (2) and (3) and our techniques from Section 6.1 for manufacturing admissible, total, and left-closed relations. First, we instantiate

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

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

$$\begin{aligned}
&p \circ h \neq \bot \wedge (\forall x_1 :: \tau_1, x_2 :: \tau_2.\ x_1 \sqsubseteq h\ x_2 \Rightarrow p\ x_1 \sqsubseteq p\ (h\ x_2))\\
&\Rightarrow ((map\ h\ l, l) \in \sqsubseteq ; lift_{[]}(\sqsubseteq ; h^{-1}) \Rightarrow (filter_{\tau_1}\ p\ (map\ h\ l),\ filter_{\tau_2}\ (p \circ h)\ l) \in \sqsubseteq ; lift_{[]}(\sqsubseteq ; h^{-1})).
\end{aligned}$$

Since $p \circ h$ is equivalent to a λ-abstraction and hence is not $\bot$, and since the second conjunct of the precondition follows from monotonicity of p, applications of laws (6) and (8) yield (2).

Second, we instantiate

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

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

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

Since the second conjunct of the precondition follows from monotonicity of p, applications of laws (7) and (9) yield (3). □

To illustrate the roles of the restrictions on p and h required in Theorem 1.1, we consider the instantiations for p, h, and l that were used in the introduction — together with the particular function definition for the sequentialized filter presented there — as counterexamples for the unrestricted equational law (1) when seq is present. The third instantiation shows that a proof of law (2) without the strictness precondition on h cannot exist (although the fourth instantiation shows that the conclusion of law (2) might happen to hold for nonstrict h). The fourth instantiation shows that strictness of h cannot be dropped from law (3). Moreover, both the third and fourth instantiations show that strictness of h cannot be dropped from law (4) either, because in both cases p is different from $\bot$ and h is total, but the equation in the conclusion of law (4) does not hold. Interestingly, p and h do not differ between these instantiations, but the small variation of l is enough for the third instantiation to make $map\ h\ (\mathit{filter}\ (p \circ h)\ l)$ ***less defined*** than $\mathit{filter}\ p\ (map\ h\ l)$, even while the fourth instantiation makes it ***more defined***. Finally, the first two instantiations show that neither the restriction that $p \neq \bot$ nor totality of h can be omitted when recovering equality.

Another illustrative take on laws (2) and (3) is to argue on an intuitive level why they hold for the sequentialized 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 (2), we need to establish that $rhs = map\ h\ (\mathit{filter}\ (p \circ h)\ l)$ is always at least as defined as $lhs = \mathit{filter}\ p\ (map\ h\ l)$ 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 $map\ h\ l$ 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 $map\ h\ \bot = \bot$. Turning to law (3), we must argue that under the additional restrictions $p \neq \bot$ and totality of h, lhs is 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 $map\ h\ l$ in lhs only encounters $\bot$ if the corresponding list element of l in rhs is itself $\bot$. Strictness of $map\ h$ thus ensures that rhs is never any more defined than lhs as a result of an application of seq to an element of $map\ h\ l$. 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 1.1 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. The proof of the inequational free theorem (2) was made possible by the asymmetry built into the new logical relation. If we were to instead replace the totality and left-closedness requirements on relational interpretations with bottom-reflectingness, and properly construct a logical relation that preserves these restrictions (which can be done), then we would only be able to obtain law (4).

8. Program transformations

Free theorems have found an important application as justifications for various kinds of program transformations for nonstrict functional languages [6, 10, 12, 32, 35]. But the presence of seq threatens the intended 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 based on the polymorphic types of arguments to the rank-2 polymorphic functions $destroy$, $build$, and $vanish_{++}$ given in Figure 5. Although we consider only list-manipulating combinators in this paper, our techniques yield similar results for analogues of them — such as those considered in [12] — which manipulate non-list data structures.

$$
\begin{aligned}
&unfoldr :: \forall\alpha\ \beta.\ (\beta \to \mathsf{Maybe}\ (\alpha, \beta)) \to \beta \to [\alpha]\\
&unfoldr\ psi\ e = \mathbf{case}\ psi\ e\ \mathbf{of}\ \mathsf{Nothing} \quad \to []\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{Just}\ (a, e') \to a : unfoldr\ psi\ e'\\
&\\
&destroy :: \forall\alpha\ \gamma.\ (\forall\beta.\ (\beta \to \mathsf{Maybe}\ (\alpha, \beta)) \to \beta \to \gamma) \to [\alpha] \to \gamma\\
&destroy\ g = g\ listpsi\ \mathbf{where}\ listpsi\ [] \qquad = \mathsf{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad listpsi\ (a : as) = \mathsf{Just}\ (a, as)\\
&\\
&foldr :: \forall\alpha\ \beta.\ (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\
&foldr\ c\ n\ [] \qquad = n\\
&foldr\ c\ n\ (a : as) = c\ a\ (foldr\ c\ n\ as)\\
&\\
&build :: \forall\alpha.\ (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]\\
&build\ g = g\ (:)\ []\\
&\\
&vanish_{++} :: \forall\alpha.\ (\forall\beta.\ \beta \to (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \beta) \to [\alpha]\\
&vanish_{++}\ g = g\ id\ (\lambda x\ h\ ys \to x : h\ ys)\ (\circ)\ []
\end{aligned}
$$

Figure 5. Functions for program transformations.

8.1. The dual of short cut fusion

Svenningsson [32] considered the $destroy/unfoldr$ rule as a dual to the $foldr/build$ rule used in short cut fusion [10] (considered in the next subsection). It can be used to eliminate intermediate lists from compositions of list producers written in terms of $unfoldr$ and list consumers written in terms of $destroy$. As shown in Sections 4 and 5 of [32], the $destroy/unfoldr$ rule overcomes some of the shortcomings short cut fusion has in eliminating intermediate lists consumed by zip-like functions, which traverse more than one data structure simultaneously, as well as by functions defined using accumulating parameters.

The function $unfoldr$ takes as input a seed value e and a step function psi which uses this seed to determine whether or not to continue unfolding the computation. If unfolding is to continue with the current seed, then the next element of $unfoldr$'s output list is returned together with a new seed; otherwise computation stops and the empty list is returned. The function $destroy$ applies a type-independent unfolding function g to the specific step function $listpsi$, which is specially tailored to uniformly consume the lists uniformly constructed by $unfoldr$. When the $destroy/unfoldr$ rule is applied, the resulting program avoids constructing intermediate lists produced by $unfoldr\ psi\ e$ and immediately consumed by $destroy\ g$. This is accomplished by applying the unfolding function g to psi and e directly.

Svenningsson describes the $destroy/unfoldr$ rule as an oriented replacement transformation, but makes no precise statement about its impact on program 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:

$$destroy\ g\ (unfoldr\ psi\ e) \sqsubseteq g\ psi\ e \tag{12}$$

Letting $g = (\lambda x\ y \to \mathbf{case}\ x\ y\ \mathbf{of}\ \mathsf{Just}\ z \to 0)$, $psi = (\lambda x \to \mathbf{if}\ x == 0\ \mathbf{then}\ \mathsf{Just}\ \bot\ \mathbf{else}\ \mathsf{Nothing})$, and $e = 0$ demonstrates that (even in the absence of seq) the equational variant of (12) does not hold in general, but Svenningsson's paper does not mention this.

While [32] 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 not taken into account. But the following two instantiations using seq break conjecture (12), making the right-hand side less defined than the left-hand side:

$$\begin{array}{lll} g = (\lambda x\ y \to seq\ x\ 0) & psi = \bot & e = 0 \\ g = (\lambda x\ y \to seq\ y\ 0) & psi = (\lambda x \to \mathsf{Nothing}) & e = \bot \end{array}$$

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

Intuitively, there are two different reasons for such decrease of definedness of $g\ psi\ e$ as compared with $destroy\ g\ (unfoldr\ psi\ e) = g\ listpsi\ (unfoldr\ psi\ e)$. First, if g applies seq on its first argument, it potentially finds $psi = \bot$ in the former but always $listpsi \neq \bot$ in the latter. Second, if g applies seq on its second argument (or another "seed" obtained from its second argument via applying its first argument repeatedly), it may find $\bot$ in the former, but a potentially non-$\bot$ value $unfoldr\ psi\ \bot$ in the latter. This depends on whether $psi\ \bot$ matches successfully against $\mathsf{Nothing}$ or $\mathsf{Just}\ (a, e')$.

To find conditions under which (12) 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 8.1. 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:

if $psi \neq \bot$ *and* $psi\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$, *then:*

$$(destroy\ g\ (unfoldr\ psi\ e), g\ psi\ e) \in \Delta^{seq}_{\tau,\emptyset} \tag{13}$$

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

$$(destroy\ g\ (unfoldr\ psi\ e), g\ psi\ e) \in (\Delta^{seq}_{\tau,\emptyset})^{-1} \tag{14}$$

Proof:
The parametricity property associated with g's type is the following instance of law (11):

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

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 $e_1 :: \tau_1$ and $e_2 :: \tau_2$ the following holds:

$$\begin{aligned}
&(g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot)\\
\wedge\ (&\quad (psi_1 \neq \bot \Rightarrow psi_2 \neq \bot)\\
&\quad \wedge (\forall b_1 :: \tau_1, b_2 :: \tau_2.\ (b_1, b_2) \in \mathcal{R} \Rightarrow (psi_1\ b_1, psi_2\ b_2) \in \sqsubseteq ; lift_{\mathsf{Maybe}}(\sqsubseteq ; lift_{(,)}(\Delta^{seq}_{\tau',[\mathcal{R}/\beta]}, \mathcal{R})))\\
\Rightarrow\ &\quad (g_{\tau_1}\ psi_1 \neq \bot \Rightarrow g_{\tau_2}\ psi_2 \neq \bot)\\
&\quad \wedge ((e_1, e_2) \in \mathcal{R} \Rightarrow (g_{\tau_1}\ psi_1\ e_1, g_{\tau_2}\ psi_2\ e_2) \in \Delta^{seq}_{\tau,[\mathcal{R}/\beta]})).
\end{aligned}$$

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 the conjuncts $g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot$ and $g_{\tau_1}\ psi_1 \neq \bot \Rightarrow g_{\tau_2}\ psi_2 \neq \bot$ from the above, and replacing the precondition $psi_1 \neq \bot \Rightarrow psi_2 \neq \bot$ with the stronger precondition $psi_2 \neq \bot$, we obtain the following weaker statement:

$$\begin{aligned}
&psi_2 \neq \bot\\
\wedge\ &(\forall b_1 :: \tau_1, b_2 :: \tau_2.\ (b_1, b_2) \in \mathcal{R} \Rightarrow (psi_1\ b_1, psi_2\ b_2) \in \sqsubseteq ; lift_{\mathsf{Maybe}}(\sqsubseteq ; lift_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R})))\\
\wedge\ &(e_1, e_2) \in \mathcal{R}\\
\Rightarrow\ &(g_{\tau_1}\ psi_1\ e_1, g_{\tau_2}\ psi_2\ e_2) \in \Delta^{seq}_{\tau,\emptyset}.
\end{aligned}$$

We consider two instantiations of this. First, we instantiate

$$\tau_1 = [\tau'], \quad \mathcal{R} = \sqsubseteq ; (unfoldr\ psi)^{-1}, \quad psi_1 = listpsi, \quad psi_2 = psi, \quad e_1 = unfoldr\ psi\ e, \quad e_2 = e$$

for $psi :: \tau_2 \to \mathsf{Maybe}\ (\tau', \tau_2)$ such that $psi\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$. Note that the instantiation for $\mathcal{R}$ is permissible because the condition on psi guarantees that $unfoldr\ psi$ is a strict function. We obtain:

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

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

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

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

$$
\begin{aligned}
\mathbf{case}\ psi\ b_2\ \mathbf{of}\ \mathsf{Nothing} &\to \mathsf{Nothing} \\
\mathsf{Just}\ (a, e') &\to \mathsf{Just}\ (a, unfoldr\ psi\ e').
\end{aligned}
$$

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

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

But this follows from reflexivity of $\sqsubseteq$, the definition of $lift_{(,)}$, law (11) for the closed type τ', and the instantiation of $\mathcal{R}$. Finally, in the case $psi\ b_2 = \mathsf{Just}\ \bot$,

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

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

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

from which law (13) follows by the definition of $destroy$.

Second, we instantiate

$$\tau_2 = [\tau'], \quad \mathcal{R} = (unfoldr\ psi)\,;\sqsubseteq, \quad psi_1 = psi, \quad psi_2 = listpsi, \quad e_1 = e, \quad e_2 = unfoldr\ psi\ e$$

for strict and total $psi :: \tau_1 \to \mathsf{Maybe}\ (\tau', \tau_1)$ 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. Of course, to guarantee that $unfoldr\ psi$ is strict and total, it is enough to require that $psi\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$, psi is total, and psi never returns $\mathsf{Just}\ \bot$ for a non-$\bot$ argument. But the argument below requires that psi *never* returns $\mathsf{Just}\ \bot$ at all, not even for the argument $\bot$. We obtain:

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

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 $b_2 :: [\tau']$, $unfoldr\ psi\ b_1 \sqsubseteq b_2$ and monotonicity of $listpsi$ imply the following inequation:

$$listpsi\ (unfoldr\ psi\ b_1) \sqsubseteq listpsi\ b_2.$$

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

$$\begin{aligned}\mathbf{case}\ psi\ b_1\ \mathbf{of}\ \mathsf{Nothing} &\to \mathsf{Nothing}\\ \mathsf{Just}\ (a, e') &\to \mathsf{Just}\ (a, unfoldr\ psi\ e').\end{aligned}$$

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 $psi\ b_1 :: \mathsf{Maybe}\ (\tau', \tau_1)$ and $listpsi\ b_2 :: \mathsf{Maybe}\ (\tau', [\tau'])$ to one of the following combinations:

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

It remains to be checked that in each of these cases the two values are related by $\sqsubseteq ; lift_{\mathsf{Maybe}}(\sqsubseteq ; lift_{(,)}(\Delta^{seq}_{\tau',\emptyset}, \mathcal{R}))$. 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 $lift_{\mathsf{Maybe}}$ and $lift_{(,)}$, law (11) 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 (14) by the definition of $destroy$. □

The laws (13) and (14) are not yet the desired inequational statements about 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 [10, 35]), 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 on the basis of Wadler's parametricity theorem [36], from which these proofs claim to be derived, but also requires Reynolds' *identity extension lemma* [28].

In addition to a standard equational semantics, Reynolds also considered an "order-relation semantics" in which the interpretations of base types are semantic approximation relations rather than identity relations. He noted that an analogue of the identity extension lemma holds for the order-relation semantics prior to the inclusion of polymorphic types. This motivates the following conjecture:

if τ is a closed type, then:

$$\Delta^{seq}_{\tau,\emptyset} = \sqsubseteq_\tau \tag{15}$$

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., $\sqsubseteq_\tau \to^{seq} \sqsubseteq_{\tau'} = \sqsubseteq_{\tau\to\tau'}$ and, e.g., $\sqsubseteq_{\mathsf{Maybe}\,\tau}\,;\, \mathit{lift}_{\mathsf{Maybe}}(\sqsubseteq_\tau) = \sqsubseteq_{\mathsf{Maybe}\,\tau}$. Thus, (15) is known to hold in most situations that are interesting in practice. For example, the value produced by $destroy\ g$ from an input list usually is not polymorphic, and, although the data structures eliminated by the program transformations studied in this paper are often polymorphic, they do not usually contain polymorphic values.

To show that conjecture (15) also holds for types involving polymorphism is more complicated. Indeed, we encounter a problem analogous to that which arises in Section 8 of [28] 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. But unless the set of functions from types to values contained by the model at polymorphic types is winnowed at construction time to exclude those which do not satisfy the required statement, the statement need not hold. Reynolds solves the analogous problem for the standard equational semantics for $\lambda^\forall$ by incorporating precisely the required statement into the definition of the set of values a polymorphic type contains; in fact, he considers the added condition to draw the dividing line between parametric and *ad hoc* polymorphism. He then argues that no values of terms expressible in the underlying language are unduly excluded by doing so by appealing to the identity extension lemma itself and to the abstraction theorem. The latter corresponds to the generalized form of Wadler's parametricity theorem in Section 6 of [36] and thus to the generalization of the fundamental property (11) for types and terms potentially containing free variables mentioned in our sketched proof in Section 6. Although no formal proofs of the analogues of (15) are presently available for either the standard equational semantics or the order-relation semantics for $\lambda^\forall$ with fixpoints and algebraic datatypes, it is reasonable to expect that any Reynolds-inspired approach to establishing these results will be immediately applicable in our setting as well.

Another approach to handling conjecture (15) is to mirror Pitts' operational techniques [25]. Pitts constructed a logical relation for the PolyFix calculus mentioned in Section 2, and proved that it interprets arbitrary closed types as contextual equivalence relations. This was used in [12, 13, 14] 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. Because we believe a Pitts-like operational approach to be the most promising one for putting parametricity results for polymorphic lambda calculi on a solid theoretical foundation, we are currently working to extend Pitts' results to accommodate seq.

Using the plausible assumption (15), the laws (13) and (14) turn into the following:

if $psi \neq \bot$ *and* $psi\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$, *then:*

$$destroy\ g\ (unfoldr\ psi\ e) \sqsubseteq g\ psi\ e \tag{16}$$

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

$$destroy\ g\ (unfoldr\ psi\ e) \sqsupseteq g\ psi\ e \tag{17}$$

In the following we argue — using counterexamples — that none of the preconditions appearing in laws (16) and (17) can be dropped. The two instantiations used to show that conjecture (12) may break in the presence of seq also show that neither $psi \neq \bot$ nor $psi\ \bot$ being $\bot$ or $\mathsf{Just}\ \bot$ would alone be enough in law (16). Likewise, each of the preconditions in law (17) really is required. The instantiations

$$\begin{array}{lll} g = (\lambda x\ y \to \mathbf{case}\ x\ \bot\ \mathbf{of}\ \mathsf{Nothing} \to 0) & psi = (\lambda x \to \mathsf{Nothing}) & e = 0 \\ g = (\lambda x\ y \to seq\ y\ 0) & psi = \bot & e = 0 \end{array}$$

(in that order) show that neither strictness of psi, nor totality of psi, can be omitted, while the instantiation immediately following conjecture (12) shows that the requirement that psi never returns $\mathsf{Just}\ \bot$ cannot be dropped. Indeed, each of these instantiations fulfills all preconditions from law (17) except for the one in question, but in all three cases $g\ psi\ e$ is strictly more defined than $destroy\ g\ (unfoldr\ psi\ e)$.

Note that the examples showing that the first and the third preconditions cannot be dropped from law (17) do not involve seq. It is also possible to give an instantiation not involving seq which shows that the potential generalization considered just after the second instantiation of (the weakening of) the parametricity property of the function argument g to $destroy$ in the proof of Theorem 8.1 would not only compromise our particular proof, but in fact cannot be proved in any manner whatsoever because the analogue of law (17) that would result does not hold. The following is such an instantiation:

$$\begin{aligned} g &= (\lambda x\ y \to \mathbf{case}\ x\ \bot\ \mathbf{of}\ \mathsf{Just}\ z \to 0) \\ psi &= (\lambda x \to \mathsf{Just}\ (\mathbf{if}\ x == 0\ \mathbf{then}\ (x, x)\ \mathbf{else}\ (x, x))) \\ e &= 0 \end{aligned}$$

On the other hand, it is not possible to construct an example without seq which shows that totality of psi is required in law (17). This is because strictness of psi and psi not returning $\mathsf{Just}\ \bot$ are shown in [14] to be sufficient to guarantee semantics-preservation of the $destroy/unfoldr$ rule in the absence of seq. When seq *is* present, conditions for semantics-preservation are obtained by combining laws (16) and (17): psi must be strict, total, distinct from $\bot$, and never return $\mathsf{Just}\ \bot$. It is interesting to note that the folkloric approach would have yielded almost the same conditions for semantics-preservation of the $destroy/unfoldr$ rule, with only the requirement that $psi \neq \bot$ left out. This requirement is, however, necessary, since the other three preconditions do not ensure that $psi \neq \bot$ if the domain of psi is $\{\bot\}$. Our approach instead gives rise to a correct equational statement, and has the added benefit of delivering inequational laws under weaker preconditions as well.

Seen from a pragmatic point of view, safeness of the $destroy/unfoldr$ transformation — in the sense of law (16) — is sufficient to justify its application even in the presence of seq. Because it ensures safeness, the requirement that the first arguments of all occurrences of $unfoldr$ in the original program are non-$\bot$ functions that return either $\bot$ or $\mathsf{Just}\ \bot$ when applied to $\bot$ therefore merits foremost consideration. Most of the examples given in [32] satisfy this restriction, with the notable exceptions of a — rather toy — definition of the empty list in terms of $unfoldr$ and the following function definition:

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

Fusion with this function as a producer can be problematic. But overall the preconditions for safeness of the $destroy/unfoldr$ rule are not terribly severe. At least it is easy to see that they are fulfilled for every producer $unfoldr\ psi\ e$ of a nonempty finite list, i.e., of a list that has at least one element and ends

with []. Detecting (automatically, in a compiler) whether a given fusion opportunity should be taken is of course a different story altogether, because it requires determining $psi \neq \bot$ and $psi\ \bot$ being $\bot$ or $\mathsf{Just}\ \bot$ statically. For the former, only sufficient conditions can be used, such as psi being an explicit λ-abstraction or a partially applied function. One of these will often be the case (and, indeed, this is so for all examples from [32]). To statically detect that $psi\ \bot \in \{\bot, \mathsf{Just}\ \bot\}$ one can resort to actually checking strictness of psi, for which various approaches are available [16, 19, 20]. In fact, one might be able to leverage strictness analysis passes that are already present in, e.g., the Glasgow Haskell Compiler.

To combine laws (16) and (17) to establish that a transformed program is *exactly* as defined as the original program, two further requirements (besides $psi \neq \bot$ and strictness of psi) on the first argument to $unfoldr$ are necessary. While totality might seem a mild restriction at first sight, even an innocently looking list producer like the running example from [32]:

$$enumFromTo\ n\ m = unfoldr\ (\lambda i \to \mathbf{if}\ i > m\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ \mathsf{Just}\ (i, i+1))\ n$$

breaks this requirement if $m = \bot$. Fortunately, this can only lead to the transformed program being more defined than the original one, not the other way around.

The condition that psi never return $\mathsf{Just}\ \bot$ in law (17) is merely an artifact of the choice of presentation of $unfoldr$ and $destroy$ using the type $\mathsf{Maybe}\ (\alpha, \beta)$ rather than a more tailored type such as the pattern functor

$$\mathbf{data}\ \mathsf{ListBase}\ \alpha\ \beta = \mathsf{N} \mid \mathsf{J}\ \alpha\ \beta$$

for lists. The small — but essential — difference is that the data constructor J is applied to values of types α and β singly, rather than encapsulated in a pair, so that $\mathsf{ListBase}\ \alpha\ \beta$ does not contain a value analogous to $\mathsf{Just}\ \bot$. The precondition that psi never returns this junk value would therefore not be needed if such a specially designed type were used. The assertion that $\mathsf{Just}\ \bot$ is "junk" here is justified by the fact that the pattern match in the definition of $unfoldr$ has only branches $\mathsf{Nothing}$ and $\mathsf{Just}\ (a, e')$, causing $\mathsf{Just}\ \bot$ to lead to a pattern match failure just as $\bot$ does. Using $\mathsf{Maybe}\ (\alpha, \beta)$ thus does not allow specification of any more list producers as instances of $unfoldr$ than using $\mathsf{ListBase}\ \alpha\ \beta$ does. This also means that not permitting psi to return $\mathsf{Just}\ \bot$ in law (17) is a rather lax restriction. Furthermore, the type $\mathsf{Maybe}\ (\alpha, \beta)$ has the advantage of being modularly constructed, with the usual attendant benefits, such as more potential for reuse. In fact, the definition of $unfoldr$ used in this paper comes straight from the Haskell 98 report [21].

8.2. Short cut fusion

The classic program transformation whose correctness is proved with a free theorem is the $foldr/build$ rule [10]. It eliminates intermediate lists from compositions of list producers written in terms of $build$ and list consumers written in terms of $foldr$ using the fact that for appropriately typed g, c, and n:

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

Definitions of $foldr$ and $build$ are given in Figure 5. The function $foldr$ takes as input a function c, a value n, and a list l, and produces a value by replacing all occurrences of $(:)$ in l by c and any occurrence of [] in l by n. For instance, $foldr\ (+)\ 0\ l$ sums the (numeric) elements of the list l. The function $build$, on the other hand, takes as input a function g providing a type-independent template for constructing "abstract" lists, and applies it to the list constructors $(:)$ and [] to get a corresponding "concrete" list.

For example, $build\ (\lambda\, c\, n \to c\ 3\ (c\ 7\ n))$ produces the list [3,7]. When law (18) is applied, the resulting program avoids constructing intermediate lists produced by $build\ g$ and immediately consumed by $foldr\ c\ n$. This is accomplished by applying g to the $(:)$- and $[]$-replacement functions c and n directly. For example, $foldr\ (+)\ 0\ (build\ (\lambda\, c\, n \to c\ 3\ (c\ 7\ n))$ is optimized to $(+)\ 3\ ((+)\ 7\ 0)$.

The following instantiations show that law (18) fails in the presence of seq:

$$\begin{array}{lll} g = seq & c = \bot & n = 0 \\ g = (\lambda\, c\, n \to seq\ (c\ \bot\ \bot)\ n) & c = (\lambda\, x\ y \to y) & n = 0 \\ g = (\lambda\, c\, n \to seq\ n\ (c\ \bot\ \bot)) & c = (:) & n = \bot \end{array}$$

In each of these cases, $g\ c\ n$ is strictly less defined than $foldr\ c\ n\ (build\ g)$. The intuitive reason for this possible decrease of definedness of the transformed program lies in the different definedness and strictness properties of the arguments supplied to g prior and after applying the $foldr/build$ rule, respectively. While the list constructors $(:)$ and $[]$ — passed to g in $build\ g$ — are both non-$\bot$, and so is any value obtained by combining them, no such *a priori* guarantees exist with respect to c and n. As a consequence, the same use of seq inside g might result in $\bot$ on the right-hand side of (18) and in a non-$\bot$ value on its left-hand side.

On the other hand, the transformed program cannot possibly be more defined than the original one, even in the presence of seq. This is proved in the following theorem, which also gives conditions guaranteeing semantics-preservation. The proofs again rely on assumption (15).

Theorem 8.2. For every closed type τ, every function

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

and appropriately typed c and n the following hold:

$$foldr\ c\ n\ (build\ g) \sqsupseteq g\ c\ n \tag{19}$$

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

$$foldr\ c\ n\ (build\ g) = g\ c\ n \tag{20}$$

Proof:
The parametricity property associated with g's type is the following instance of law (11):

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

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 $c_1 :: \tau \to \tau_1 \to \tau_1$ and

$c_2 :: \tau \to \tau_2 \to \tau_2$, and values $n_1 :: \tau_1$ and $n_2 :: \tau_2$ the following holds:

$$\begin{aligned}
&(g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot) \\
\wedge\ (&\quad (c_1 \neq \bot \Rightarrow c_2 \neq \bot) \\
&\wedge (\forall x_1, x_2 :: \tau, y_1 :: \tau_1, y_2 :: \tau_2. \\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,[\mathcal{R}/\beta]} \Rightarrow \quad (c_1\ x_1 \neq \bot \Rightarrow c_2\ x_2 \neq \bot) \\
&\qquad\qquad \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (c_1\ x_1\ y_1, c_2\ x_2\ y_2) \in \mathcal{R})) \\
&\Rightarrow \quad (g_{\tau_1}\ c_1 \neq \bot \Rightarrow g_{\tau_2}\ c_2 \neq \bot) \\
&\wedge ((n_1, n_2) \in \mathcal{R} \Rightarrow (g_{\tau_1}\ c_1\ n_1, g_{\tau_2}\ c_2\ n_2) \in \mathcal{R})).
\end{aligned}$$

Using the fact that for the closed type τ we have $\Delta^{seq}_{\tau,[\mathcal{R}/\beta]} = \Delta^{seq}_{\tau,\emptyset}$, dropping the conjuncts $g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot$ and $g_{\tau_1}\ c_1 \neq \bot \Rightarrow g_{\tau_2}\ c_2 \neq \bot$ from the above, and strengthening two preconditions by replacing $c_1 \neq \bot \Rightarrow c_2 \neq \bot$ and $c_1\ x_1 \neq \bot \Rightarrow c_2\ x_2 \neq \bot$ with $c_2 \neq \bot$ and $c_2\ x_2 \neq \bot$, respectively, we obtain the following weaker statement:

$$\begin{aligned}
&c_2 \neq \bot \\
\wedge\ &(\forall x_1, x_2 :: \tau, y_1 :: \tau_1, y_2 :: \tau_2. \\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset} \Rightarrow c_2\ x_2 \neq \bot \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (c_1\ x_1\ y_1, c_2\ x_2\ y_2) \in \mathcal{R})) \\
\wedge\ &(n_1, n_2) \in \mathcal{R} \\
\Rightarrow\ &(g_{\tau_1}\ c_1\ n_1, g_{\tau_2}\ c_2\ n_2) \in \mathcal{R}.
\end{aligned}$$

We consider two instantiations of this. First, we instantiate as follows:

$$\tau_2 = [\tau], \quad \mathcal{R} = \sqsubseteq ; (\mathit{foldr}\ c\ n)^{-1}, \quad c_1 = c, \quad c_2 = (:), \quad n_1 = n, \quad n_2 = [].$$

Note that the instantiation for $\mathcal{R}$ is permissible because $\mathit{foldr}\ c\ n$ is always a strict function. We obtain:

$$\begin{aligned}
&(:) \neq \bot \\
\wedge\ &(\forall x_1, x_2 :: \tau, y_1 :: \tau_1, y_2 :: [\tau]. \\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset} \Rightarrow (:)\ x_2 \neq \bot \wedge (y_1 \sqsubseteq \mathit{foldr}\ c\ n\ y_2 \Rightarrow c\ x_1\ y_1 \sqsubseteq \mathit{foldr}\ c\ n\ (x_2 : y_2))) \\
\wedge\ &n \sqsubseteq \mathit{foldr}\ c\ n\ [] \\
\Rightarrow\ &g_{\tau_1}\ c\ n \sqsubseteq \mathit{foldr}\ c\ n\ (g_{[\tau]}\ (:)\ []).
\end{aligned}$$

The first conjunct of the precondition is trivially true; the third one follows from the definition of foldr and the reflexivity of $\sqsubseteq$. Since an application of $(:)$ to only one argument is partial and hence is not $\bot$, we can prove that the second conjunct of the precondition holds by showing that $(x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset}$ and $y_1 \sqsubseteq \mathit{foldr}\ c\ n\ y_2$ imply

$$c\ x_1\ y_1 \sqsubseteq \mathit{foldr}\ c\ n\ (x_2 : y_2).$$

Since by the definition of foldr the right-hand side of this inequation is equal to $c\ x_2\ (\mathit{foldr}\ c\ n\ y_2)$, monotonicity of c gives the desired statement provided that $x_1 \sqsubseteq x_2$, which follows from $(x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset}$ by assumption (15). Having established the validity of all three conjuncts of the precondition in the above implication, its conclusion gives law (19) by the definition of $build$.

Second, we instantiate

$$\tau_1 = [\tau], \quad \mathcal{R} = (\mathit{foldr}\ c\ n)\,;\sqsubseteq, \quad c_1 = (:), \quad c_2 = c, \quad n_1 = [], \quad n_2 = n$$

for $c :: \tau \to \tau_2 \to \tau_2$ and $n :: \tau_2$ such that $c \perp \perp \neq \perp$ and $n \neq \perp$. The instantiation for $\mathcal{R}$ is permissible because the conditions on c and n guarantee that $\mathit{foldr}\ c\ n$ is a strict and total function. We obtain:

$$\begin{aligned}
&c \neq \perp \\
\wedge\ &(\forall x_1, x_2 :: \tau, y_1 :: [\tau], y_2 :: \tau_2. \\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset} \Rightarrow c\ x_2 \neq \perp \wedge (\mathit{foldr}\ c\ n\ y_1 \sqsubseteq y_2 \Rightarrow \mathit{foldr}\ c\ n\ (x_1 : y_1) \sqsubseteq c\ x_2\ y_2)) \\
\wedge\ &\mathit{foldr}\ c\ n\ [] \sqsubseteq n \\
\Rightarrow\ &\mathit{foldr}\ c\ n\ (g_{[\tau]}\ (:)\ []) \sqsubseteq g_{\tau_2}\ c\ n.
\end{aligned}$$

The first conjunct of the precondition follows from $c \perp \perp \neq \perp$; the third one follows from the definition of foldr and the reflexivity of $\sqsubseteq$. Since $c \perp \perp \neq \perp$ also implies that $c\ x_2 \neq \perp$ for every $x_2 :: \tau$, we can prove that the second conjunct of the precondition holds by showing that $(x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset}$ and $\mathit{foldr}\ c\ n\ y_1 \sqsubseteq y_2$ imply

$$\mathit{foldr}\ c\ n\ (x_1 : y_1) \sqsubseteq c\ x_2\ y_2.$$

This is established via reasoning similar to that used above, using the definition of foldr, monotonicity of c, and assumption (15). The conclusion of the above implication, together with the definition of $build$ and the previously proven (19), then gives (20). □

Law (19) gives only partial correctness of the $\mathit{foldr}/\mathit{build}$ rule in general because the transformed program may be less defined than the original one. But that it holds without preconditions, even in the presence of seq, was not previously known. To recover total correctness in law (20), c and n must be restricted so that $\mathit{foldr}\ c\ n$ is total (in addition to being strict, which it always is by the definition of foldr). That the precondition on c cannot be dropped from law (20) can be seen by considering the first two instantiations preceding Theorem 8.2. The third instantiation shows that the precondition on n cannot be dropped either. That totality of $\mathit{foldr}\ c\ n$ is needed to guarantee total correctness in the presence of seq just happens to coincide with what the folkloric approach had to say about the $\mathit{foldr}/\mathit{build}$ rule; indeed, this coincidence is likely responsible for its failure having gone unnoticed for so long.

It should be noted that the restrictions on c and n ensuring totality of $\mathit{foldr}\ c\ n$ can actually hinder fusion opportunities; in fact, the requirements that $c \perp \perp \neq \perp$ and $n \neq \perp$ are *equivalent* to totality of $\mathit{foldr}\ c\ n$. While many functions consuming lists by structure — e.g., $\mathit{map}\ h = \mathit{foldr}\ (\lambda x\ ys \to h\ x : ys)\ []$ — do indeed map non-$\perp$ lists to non-$\perp$ results, other functions expressed in terms of foldr — even ones from Haskell's standard prelude — break the required preconditions. A familiar example of a function of the latter kind is the function $\mathit{sum} = \mathit{foldr}\ (+)\ 0$ discussed at the start of this subsection. As with the preconditions of law (16) in the previous subsection, only sufficient criteria can be used when verifying inside a compiler that a given instantiation fulfills $c \perp \perp \neq \perp$ and $n \neq \perp$.

As a final interesting observation, note that seq compromises the duality between the $\mathit{foldr}/\mathit{build}$ and $\mathit{destroy}/\mathit{unfoldr}$ rules exhibited in its absence. For instance, the $\mathit{foldr}/\mathit{build}$ rule can never produce a program that is strictly more defined than the original (cf. law (19)), whereas we have given examples (in Section 8.1) showing that the $\mathit{destroy}/\mathit{unfoldr}$ rule can produce both programs which are strictly less defined, and programs which are strictly more defined, than those from which they came.

8.3. The concatenate vanishes for free

In [35] the function $vanish_{+\!+}$ from Figure 5 was given, together with a proof of the following law for appropriately typed g in the absence of seq:

$$g\ []\ (:)\ (+\!+) = vanish_{+\!+}\ g \tag{21}$$

Read from left to right, this law can be considered as a program transformation rule that eliminates concatenate operations from uniformly abstracted list producers. Examples of its use to improve the performance of list-manipulating programs involving calls to $+\!+$ — some of which cannot be optimized by other known techniques — are given in [35].

In Appendix B of [35] it was noted that in the presence of seq the transformation might improve the termination behavior of programs. This is witnessed by the following two instantiations, each of which makes the right-hand side of (21) more defined than the left-hand side:

$$g = (\lambda\, n\ c\ a \to seq\ (n\ `a`\ \bot)\ n)$$
$$g = (\lambda\, n\ c\ a \to seq\ (\bot\ `a`\ (0\ `c`\ n))\ n)$$

The intuitive reason for the possible increase of definedness of the transformed program (as well as for why the definedness cannot possibly be decreased) lies in the different strictness properties of the concatenate operation $(+\!+)$, on the one hand, and its abstract replacement $(\circ)$ in the efficient list implementation encapsulated by $vanish_{+\!+}$, on the other. While $(\circ)$ applied to two arbitrary arguments always returns a non-$\bot$ value — namely, a λ-abstraction — applications of $(+\!+)$ yield $\bot$ if the first argument is $[]$ and the second one is $\bot$, or if the first argument is $\bot$ and the second one is arbitrary. As a result, the same use of seq inside g might result in $\bot$ on the left-hand side of (21) and in a non-$\bot$ value on its right-hand side.

The sketched proof in [35] that a converse failure of law (21) indeed cannot occur anticipated some of the ideas from the present paper. In particular, it used an asymmetric approach, although it 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 (11), and assumption (15), we now more rigorously obtain the following (as well as analogous inequational laws for other $vanish$-combinators given in [35]).

Theorem 8.3. For every closed type τ and every function

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

the following holds:

$$g\ []\ (:)\ (+\!+) \sqsubseteq vanish_{+\!+}\ g \tag{22}$$

Proof:
The parametricity property associated with g's type is the following instance of law (11):

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

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 $\mathcal{R} \in Rel^{seq}(\tau_1, \tau_2)$, values $n_1 :: \tau_1$ and $n_2 :: \tau_2$, and functions $c_1 :: \tau \to \tau_1 \to \tau_1$, $c_2 :: \tau \to \tau_2 \to \tau_2$, $a_1 :: \tau_1 \to \tau_1 \to \tau_1$, and $a_2 :: \tau_2 \to \tau_2 \to \tau_2$ the following holds:

$$
\begin{aligned}
&(g_{\tau_1} \neq \bot \Rightarrow g_{\tau_2} \neq \bot) \\
\wedge\ &((n_1, n_2) \in \mathcal{R} \Rightarrow \quad (g_{\tau_1}\ n_1 \neq \bot \Rightarrow g_{\tau_2}\ n_2 \neq \bot) \\
&\qquad\qquad \wedge (\quad (c_1 \neq \bot \Rightarrow c_2 \neq \bot) \\
&\qquad\qquad\quad \wedge (\forall x_1, x_2 :: \tau, y_1 :: \tau_1, y_2 :: \tau_2. \\
&\qquad\qquad\qquad (x_1, x_2) \in \Delta^{seq}_{\tau,[\mathcal{R}/\beta]} \Rightarrow \quad (c_1\ x_1 \neq \bot \Rightarrow c_2\ x_2 \neq \bot) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (c_1\ x_1\ y_1, c_2\ x_2\ y_2) \in \mathcal{R})) \\
&\qquad\qquad \Rightarrow \quad (g_{\tau_1}\ n_1\ c_1 \neq \bot \Rightarrow g_{\tau_2}\ n_2\ c_2 \neq \bot) \\
&\qquad\qquad\quad \wedge (\quad (a_1 \neq \bot \Rightarrow a_2 \neq \bot) \\
&\qquad\qquad\qquad \wedge (\forall x_1, y_1 :: \tau_1, x_2, y_2 :: \tau_2. \\
&\qquad\qquad\qquad\quad (x_1, x_2) \in \mathcal{R} \Rightarrow \quad (a_1\ x_1 \neq \bot \Rightarrow a_2\ x_2 \neq \bot) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (a_1\ x_1\ y_1, a_2\ x_2\ y_2) \in \mathcal{R})) \\
&\qquad\qquad\qquad \Rightarrow (g_{\tau_1}\ n_1\ c_1\ a_1, g_{\tau_2}\ n_2\ c_2\ a_2) \in \mathcal{R}))).
\end{aligned}
$$

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

$$
\begin{aligned}
&(n_1, n_2) \in \mathcal{R} \\
\wedge\ &c_2 \neq \bot \\
\wedge\ &(\forall x_1, x_2 :: \tau, y_1 :: \tau_1, y_2 :: \tau_2. \\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset} \Rightarrow c_2\ x_2 \neq \bot \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (c_1\ x_1\ y_1, c_2\ x_2\ y_2) \in \mathcal{R})) \\
\wedge\ &a_2 \neq \bot \\
\wedge\ &(\forall x_1, y_1 :: \tau_1, x_2, y_2 :: \tau_2. \\
&\quad (x_1, x_2) \in \mathcal{R} \Rightarrow a_2\ x_2 \neq \bot \wedge ((y_1, y_2) \in \mathcal{R} \Rightarrow (a_1\ x_1\ y_1, a_2\ x_2\ y_2) \in \mathcal{R})) \\
\Rightarrow\ &(g_{\tau_1}\ n_1\ c_1\ a_1, g_{\tau_2}\ n_2\ c_2\ a_2) \in \mathcal{R}.
\end{aligned}
$$

We instantiate as follows:

$$
\begin{aligned}
&\tau_1 = [\tau], \quad \tau_2 = [\tau] \to [\tau], \quad \mathcal{R} = \{(p, q) \mid \forall u :: [\tau].\ p +\!\!+ u \sqsubseteq q\ u\}, \quad n_1 = [], \\
&n_2 = id, \quad c_1 = (:), \qquad c_2 = (\lambda x\ h\ ys \to x : h\ ys), \qquad a_1 = (+\!\!+), \quad a_2 = (\circ).
\end{aligned}
$$

Permissibility of the instantiation for $\mathcal{R}$ follows from its equivalence to $(\lambda p \to seq\ p\ (p +\!\!+))\,; \sqsubseteq$, which

is easily established. We obtain (modulo some beta-reductions):

$$
\begin{aligned}
&(\forall u :: [\tau].\ [] +\!\!+ u \sqsubseteq id\ u)\\
\wedge\ &(\lambda x\ h\ ys \to x : h\ ys) \neq \bot\\
\wedge\ &(\forall x_1, x_2 :: \tau, y_1 :: [\tau], y_2 :: [\tau] \to [\tau].\\
&\quad (x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset} \Rightarrow (\lambda h\ ys \to x_2 : h\ ys) \neq \bot \wedge (\quad (\forall u :: [\tau].\ y_1 +\!\!+ u \sqsubseteq y_2\ u)\\
&\qquad\qquad \Rightarrow (\forall u :: [\tau].\ (x_1 : y_1) +\!\!+ u \sqsubseteq x_2 : y_2\ u)))\\
\wedge\ &(\circ) \neq \bot\\
\wedge\ &(\forall x_1, y_1 :: [\tau], x_2, y_2 :: [\tau] \to [\tau].\\
&\quad (\forall u :: [\tau].\ x_1 +\!\!+ u \sqsubseteq x_2\ u) \Rightarrow (\circ)\ x_2 \neq \bot \wedge (\quad (\forall u :: [\tau].\ y_1 +\!\!+ u \sqsubseteq y_2\ u)\\
&\qquad\qquad \Rightarrow (\forall u :: [\tau].\ (x_1 +\!\!+ y_1) +\!\!+ u \sqsubseteq (x_2 \circ y_2)\ u)))\\
\Rightarrow\ &(\forall u :: [\tau].\ (g_{[\tau]}\ []\ (:)\ (+\!\!+)) +\!\!+ u \sqsubseteq g_{[\tau]\to[\tau]}\ id\ (\lambda x\ h\ ys \to x : h\ ys)\ (\circ)\ u).
\end{aligned}
$$

The first conjunct of the precondition follows from the definitions of $(+\!\!+)$ and id and the reflexivity of $\sqsubseteq$. Since a λ-abstraction is always distinct from $\bot$, the second conjunct of the precondition is fulfilled, and we can prove the third one by showing that $(x_1, x_2) \in \Delta^{seq}_{\tau,\emptyset}$ and $(\forall u :: [\tau].\ y_1 +\!\!+ u \sqsubseteq y_2\ u)$ imply

$$\forall u :: [\tau].\ (x_1 : y_1) +\!\!+ u \sqsubseteq x_2 : y_2\ u.$$

Using the definition of $(+\!\!+)$, this follows from assumption (15) and the monotonicity of $(:)$. Since $(\circ)$, as well as an application of it to only one argument, is distinct from $\bot$, the fourth conjunct of the precondition in the above implication is fulfilled, and we can prove the fifth one by showing that $(\forall u :: [\tau].\ x_1 +\!\!+ u \sqsubseteq x_2\ u)$ and $(\forall u :: [\tau].\ y_1 +\!\!+ u \sqsubseteq y_2\ u)$ imply

$$\forall u :: [\tau].\ (x_1 +\!\!+ y_1) +\!\!+ u \sqsubseteq (x_2 \circ y_2)\ u.$$

Using the definition of $(\circ)$, this follows from the associativity and monotonicity of $(+\!\!+)$, and the transitivity of $\sqsubseteq$. Having established the validity of all five conjuncts of the precondition, we can use the conclusion of the above implication for $u = []$ to obtain

$$(g_{[\tau]}\ []\ (:)\ (+\!\!+)) +\!\!+ [] \sqsubseteq g_{[\tau]\to[\tau]}\ id\ (\lambda x\ h\ ys \to x : h\ ys)\ (\circ)\ [],$$

from which law (22) follows by the definition of $vanish_{+\!\!+}$ and the fact that $[]$ is a right unit for $(+\!\!+)$. □

In contrast to the situation for Theorems 1.1, 8.1, and 8.2, we do not provide a second law in Theorem 8.3, obtained from a "dual" instantiation of the parametricity property derived in its proof. The reason is that for the instantiation of $\mathcal{R}$ to

$$\sqsubseteq ; (\lambda p \to seq\ p\ (p +\!\!+))^{-1} = \{(q, p) \mid \forall u :: [\tau].\ q\ u \sqsubseteq p +\!\!+ u\} \setminus \{((\lambda u \to \bot), \bot)\}$$

it is not possible to show that $(x_1, x_2) \in \mathcal{R}$ and $(y_1, y_2) \in \mathcal{R}$ imply $(a_1\ x_1\ y_1, a_2\ x_2\ y_2) \in \mathcal{R}$ for $a_1 = (\circ)$ and $a_2 = (+\!\!+)$. This is because $(\bot, \bot) \in \mathcal{R}$, but $(\bot \circ \bot, \bot +\!\!+ \bot) = ((\lambda u \to \bot), \bot) \notin \mathcal{R}$. On the other hand, dropping the exclusion of $((\lambda u \to \bot), \bot)$ from $\mathcal{R}$ is not an option because then the relation would not be total, as is required. Recall that the counterexamples to the equational law (21)

from the beginning of this subsection emerged exactly from the different propagation behaviors of $(\circ)$ and $(+\!\!+)$ with respect to $\bot$ arguments.

Unlike the program transformations considered in the previous two subsections, the rule for $vanish_{+\!\!+}$ is not parametrized over values other than g which could somehow be constrained. Since putting any preconditions on g itself is not possible if the only information we want to use about it is its type, there is clearly no way to recover semantics-preservation of the $vanish_{+\!\!+}$ rule in the presence of seq. Hence, a purely equational approach to free theorems would again be strictly inferior to ours here, which recovers at least safeness of the transformation.

9. Conclusions and 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 — and the program transformations based on them — in a nonstrict functional language. We have shown that the conventional wisdom regarding free theorems in Haskell with seq is incorrect, i.e., that quantifying only over admissible and bottom-reflecting relations in the $\forall$-clause of the standard logical relation is *not* enough to recover from the failure, in the presence of seq, of the standard (equational) free theorems derived from it. By addressing the subtle issues which arise when propagating up the type hierarchy restrictions imposed on a logical relation in order to accommodate seq, we have provided a new logical relation whose parametricity theorem allows the derivation of free theorems that remain valid even when seq is present. A crucial ingredient of our approach is the use of an asymmetric logical relation, which leads to "inequational" versions of the standard free theorems — enriched by preconditions guaranteeing their validity — for the subset of Haskell corresponding to $\lambda^{\forall}$ with fixpoints, algebraic datatypes, and seq. Preconditions for equational free theorems which hold even in the presence of seq can be obtained by combining those for their two corresponding inequational variants. But the inequational approach has value in its own right as well. As we have shown here, it allows the derivation of more free theorems than would be possible via a purely equational revision of the standard approach. It also provides a means of performing more detailed analyses of the impact of seq on free theorems-based program transformations than is possible if we insist that the semantics of programs remain completely unchanged by the transformations.

It has been suggested that simply checking whether or not a function uses seq might provide a simpler alternative to verifying the kind of preconditions we derive for our free theorems. To functions that do not use seq we can then safely apply transformations based on standard free theorems. But for those that do, such an analysis will only allow us to deduce that the standard transformations might fail. The goal of the work reported here is to be able to say something more in the latter case by understanding the conditions under which free theorems-based program transformations can safely be applied even when seq is present.

The *rule pragmas* of [24] have successfully been used to incorporate standard free theorems-based transformation rules into the Glasgow Haskell Compiler. The same approach could be used to implement the transformation rules developed in this paper, assuming the development of "conditional" rule pragmas that allow application of rules only when certain preconditions — such as those accompanying our free theorems-based transformation rules — are met.

To ensure that seq impacts statements about programs only insofar as those programs actually use it, a qualified type system along the lines of [17] could be devised and employed in derivations of free

theorems. 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. If that challenge can be met, then the multi-parameter type class extension of Haskell [22] might enable a very fine-grained analysis of the preconditions necessary for free theorems to hold. By imposing different (from one another) restrictions on the relational interpretations of the two types over which seq is polymorphic, this extension could be used to account for the distinct roles these relations play in seq's parametricity property, as well as their interplay.

Although free theorems derived from our new logical relation hold for programs which do not contain seq at all, they may — of course — 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 also prove worthwhile in that setting. For example, the strongest justification for the $destroy/unfoldr$ rule in a nonstrict language without seq that could be proved in [14] was semantics-preservation for strict psi that never returns Just $\bot$. But by employing the asymmetry idea it should also be possible to establish the inequational law (12) without preconditions.

The topic of free theorems for purely strict languages has been governed by the same conventional wisdom as that for nonstrict languages which include a polymorphic strictness primitive. This conventional wisdom yields, for example, that in a strict language every function $filter$ with the type given in the introduction fulfills the free theorem $(filter\ (p \circ h)\ [], filter\ p\ []) \in lift_{[]}(h)$ for appropriately typed p and strict and total h. But this is not necessarily true if $h = \bot$ — which could still be total because its type could be of the form $(\forall\alpha.\,\alpha) \to \cdots$, in which case no non-$\bot$ arguments are possible — because under strict semantics we would have $filter\ (p \circ h)\ [] = \bot$, whereas $filter\ p\ []$ could be different from $\bot$, in contradiction to the definition of $lift_{[]}(h)$. It would therefore be interesting to investigate what our approach has to say about free theorems in pure functional languages which have the type structure considered in this paper but a strict evaluation order. Clearly, every program in such a language can be type-preservingly translated — using seq to explicitly force the evaluation of function and constructor arguments — into an equivalent program in a language for which our logical relation yields valid parametricity properties. This suggests that those parametricity properties also hold for the strict language. However, it does *not* mean that the free theorems obtained by instantiating such parametricity properties will always be the same in the strict setting as in the nonstrict setting with seq. For example, to establish the equality stated in law (1) from the parametricity property derived from $filter$'s type in the proof of Theorem 1.1, the key condition — besides strictness and totality of h — was that $p \circ h \neq \bot \Leftrightarrow p \neq \bot$. While in a nonstrict language including seq this condition is equivalent to $p \neq \bot$, in a strict language it is equivalent to $p = \bot \vee h \neq \bot$. But since in a strict language law (1) obviously holds for $h = \bot$ as well, we could completely do away with all preconditions other than strictness and totality of h here. Indeed, the first counterexample from the introduction does not break law (1) in a purely strict language. That some preconditions imposed in the setting of a nonstrict language including seq might be superfluous in a purely strict language is not entirely surprising, given that a program which is strict everywhere behaves in a more disciplined fashion than one which mixes strictness and nonstrictness at will. But, perhaps unexpectedly, the reverse situation can also arise: $(filter\ (p \circ h)\ [], filter\ p\ []) \in lift_{[]}(h)$ is a free theorem in the setting of the present paper for $p \neq \bot$ and strict and total h, but these conditions are not sufficient in a purely strict language (as seen above). In addition to such differences that may arise when *instantiating* properties derived from types to obtain free theorems in different settings, it is also conceivable that in a purely strict language the parametricity properties *themselves* could be strictly less restrictive than those in the setting of the present paper. There is much room for future research here,

especially if one also considers functional languages in which strict evaluation is the default but laziness can be imposed using special annotations (somewhat dually to the use of seq for introducing strictness; cf. [30], for example).

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 [25]. 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. A starting point for such an investigation could be Abadi's study [5] of the connection between $\top\top$-closed relations (as used by Pitts in the presence of fixpoints, but absence of seq) and admissibility.

Acknowledgments

We would like to thank the anonymous referees for their useful suggestions regarding the presentation of the paper.

A. Continuity of relational interpretations of datatypes

We show that for all left-closed and continuous relations $\mathcal{R}$ and $\mathcal{S}$ the relations $\sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$, $\sqsubseteq ; \mathit{lift}_{(,)}(\mathcal{R}, \mathcal{S})$, and $\sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R})$ are continuous. Similar proofs are possible for other algebraic datatypes.

A.1. Maybe

Let $\mathcal{R} \in Rel(\tau_1, \tau_2)$ be a left-closed and continuous relation. We prove the continuity of

$$\sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R}) \in Rel(\mathsf{Maybe}\ \tau_1, \mathsf{Maybe}\ \tau_2)$$

from the continuity of $\sqsubseteq$ and $\mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$ as follows.

The following table lists all the possible combinations of values x and y such that $(x, y) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$. For each of these cases it also defines a value $f_{x,y} :: \mathsf{Maybe}\ \tau_1$.

x	y	$\mid (x, y) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$	$f_{x,y} :: \mathsf{Maybe}\ \tau_1$
$\bot$	$\bot$		$\bot$
$\bot$	Nothing		Nothing
Nothing	Nothing		Nothing
$\bot$	Just b	$\mid \exists a.\ (a, b) \in \mathcal{R}$	Just $\bot$
Just a	Just b	$\mid (a, b) \in \sqsubseteq ; \mathcal{R}$	Just a

The idea is to define $f_{x,y}$ such that it inherits the structure of y, but uses data entries from x where available. It is easy to check that for every $(x, y), (x', y') \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$ such that $x \sqsubseteq x'$ and

$y \sqsubseteq y'$, we have $f_{x,y} \sqsubseteq f_{x',y'}$. This could also be deduced from monotonicity of all functions definable in Haskell because in fact $f_{x,y}$ can be computed as

$$\begin{aligned}&\mathbf{case}\ y\ \mathbf{of}\ \mathsf{Nothing} \to \mathsf{Nothing}\\ &\qquad\qquad\ \mathsf{Just}\ b \quad \to \mathsf{Just}\ (\mathbf{case}\ x\ \mathbf{of}\ \mathsf{Just}\ a \to a).\end{aligned}$$

Moreover, for all the above cases it is easy to check that $x \sqsubseteq f_{x,y}$ holds and, since $\mathcal{R}$ is left-closed, that $(f_{x,y}, y) \in \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$ holds as well.

Now, consider chains $x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ and $y_1 \sqsubseteq y_2 \sqsubseteq \ldots$, where for every i, $(x_i, y_i) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$. The above observations imply that $f_{x_1,y_1} \sqsubseteq f_{x_2,y_2} \sqsubseteq \ldots$ and that for every i we have $x_i \sqsubseteq f_{x_i,y_i}$ and $(f_{x_i,y_i}, y_i) \in \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$. Since $\sqsubseteq$ and $\mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$ are continuous, this implies that $\bigsqcup x_i \sqsubseteq \bigsqcup f_{x_i,y_i}$ and $(\bigsqcup f_{x_i,y_i}, \bigsqcup y_i) \in \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R})$, and hence

$$(\bigsqcup x_i, \bigsqcup y_i) \in \sqsubseteq ; \mathit{lift}_{\mathsf{Maybe}}(\mathcal{R}).$$

A.2. Pairs

Let $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$ and $\mathcal{S} \in \mathit{Rel}(\tau_1', \tau_2')$ be left-closed and continuous relations. The continuity of

$$\sqsubseteq ; \mathit{lift}_{(,)}(\mathcal{R}, \mathcal{S}) \in \mathit{Rel}((\tau_1, \tau_1'), (\tau_2, \tau_2'))$$

is proved from the continuity of $\sqsubseteq$ and $\mathit{lift}_{(,)}(\mathcal{R}, \mathcal{S})$ using the technique developed in Section A.1, but using the following table:

x	y	$\mid (x,y) \in \sqsubseteq ; \mathit{lift}_{(,)}(\mathcal{R}, \mathcal{S})$	$f_{x,y} :: (\tau_1, \tau_1')$
$\bot$	$\bot$		$\bot$
$\bot$	(b, b')	$\mid \exists a, a'.\ (a,b) \in \mathcal{R} \wedge (a', b') \in \mathcal{S}$	$(\bot, \bot)$
(a, a')	(b, b')	$\mid (a,b) \in \sqsubseteq ; \mathcal{R} \wedge (a', b') \in \sqsubseteq ; \mathcal{S}$	(a, a')

A.3. Lists

Let $\mathcal{R} \in \mathit{Rel}(\tau_1, \tau_2)$ be a left-closed and continuous relation. The continuity of

$$\sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R}) \in \mathit{Rel}([\tau_1], [\tau_2])$$

is proved from the continuity of $\sqsubseteq$ and $\mathit{lift}_{[]}(\mathcal{R})$ using the technique developed in Section A.1, but using the following table:

x	y	$\mid (x,y) \in \sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R})$	$f_{x,y} :: [\tau_1]$
$\bot$	$\bot$		$\bot$
$\bot$	$[]$		$[]$
$[]$	$[]$		$[]$
$\bot$	$b : b'$	$\mid \exists a, a'.\ (a,b) \in \mathcal{R} \wedge (a', b') \in \mathit{lift}_{[]}(\mathcal{R})$	$\bot : f_{\bot, b'}$
$a : a'$	$b : b'$	$\mid (a,b) \in \sqsubseteq ; \mathcal{R} \wedge (a', b') \in \sqsubseteq ; \mathit{lift}_{[]}(\mathcal{R})$	$a : f_{a', b'}$

Note that the definition of $f_{x,y}$ has a coinductive flavor since $\mathit{lift}_{[]}(\mathcal{R})$ is defined as a greatest fixpoint.

References

[1] The Program Transformation Wiki (`http://www.program-transformation.org/Transform`).

[2] `http://wwwtcs.inf.tu-dresden.de/~voigt/Seq.lhs`.

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

[4] Special Issue on Program Transformation, *Science of Computer Programming*, **52**, 2004, 1–371.

[5] Abadi, M.: $\top\top$-closed relations and admissibility, *Mathematical Structures in Computer Science*, **10**, 2000, 313–320.

[6] Chitil, O.: Type inference builds a short cut to deforestation, *International Conference on Functional Programming, Proceedings*, pages 249–260, ACM Press, 1999.

[7] Cosmadakis, S., Meyer, A., Riecke, J.: Completeness for typed lazy inequalities, *Logic in Computer Science, Proceedings*, pages 312–320, IEEE Computer Society Press, 1990.

[8] Czarnecki, K., Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[9] Friedman, H.: Equality between functionals, *Logic Colloquium '72–73, Proceedings*, pages 22–37, Springer-Verlag, 1975.

[10] Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232, ACM Press, 1993.

[11] Girard, J.-Y.: *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*, Ph.D. Thesis, Université Paris VII, 1972.

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

[13] Johann, P.: Short cut fusion is correct, *Journal of Functional Programming*, **13**, 2003, 797–814.

[14] Johann, P.: On proving the correctness of program transformations based on free theorems for higher-order polymorphic calculi, *Mathematical Structures in Computer Science*, **15**, 2005, 201–229.

[15] Johann, P., Voigtländer, J.: Free theorems in the presence of *seq*, *Principles of Programming Languages, Proceedings*, pages 99–110, ACM Press, 2004.

[16] Kuo, T.-M., Mishra, P.: Strictness analysis: A new perspective based on type inference, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 260–272, ACM Press, 1989.

[17] Launchbury, J., Paterson, R.: Parametricity and unboxing with unpointed types, *European Symposium on Programming, Proceedings*, pages 204–218, Springer-Verlag, 1996.

[18] Leivant, D.: Polymorphic type inference, *Principles of Programming Languages, Proceedings*, pages 88–98, ACM Press, 1983.

[19] Mycroft, A.: *Abstract Interpretation and Optimising Transformations for Applicative Programs*, Ph.D. Thesis, University of Edinburgh, 1981.

[20] Nöcker, E.: Strictness analysis using abstract reduction, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 255–265, ACM Press, 1993.

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

[22] Peyton Jones, S., Jones, M., Meijer, E.: Type classes: An exploration of the design space, *Haskell Workshop, Proceedings*, 1997.

[23] Peyton Jones, S., Launchbury, J., Shields, M., Tolmach, A.: Bridging the gulf: A common intermediate language for ML and Haskell, *Principles of Programming Languages, Proceedings*, pages 49–61, ACM Press, 1998.

[24] Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: Rewriting as a practical optimisation technique in GHC, *Haskell Workshop, Proceedings*, pages 203–233, 2001.

[25] Pitts, A.: Parametric polymorphism and operational equivalence, *Mathematical Structures in Computer Science*, **10**, 2000, 321–359.

[26] Plotkin, G.: Lambda-definability in the full type hierarchy, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373, Academic Press, 1980.

[27] Reynolds, J.: Towards a theory of type structure, *Colloque sur la Programmation, Proceedings*, pages 408–423, Springer-Verlag, 1974.

[28] Reynolds, J.: Types, abstraction and parametric polymorphism, *Information Processing, Proceedings*, pages 513–523, Elsevier Science Publishers B.V., 1983.

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

[30] Sheard, T.: A pure language with default strict evaluation order and explicit laziness, *presented at the Haskell Workshop*, 2003, available from `http://www.cs.pdx.edu/~sheard/papers/ExplicitLazy.ps`.

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

[32] Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions, *International Conference on Functional Programming, Proceedings*, pages 124–132, ACM Press, 2002.

[33] Takano, A., Meijer, E.: Shortcut deforestation in calculational form, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 306–313, ACM Press, 1995.

[34] Trinder, P., Hammond, K., Loidl, H.-W., Peyton Jones, S.: Algorithm + Strategy = Parallelism, *Journal of Functional Programming*, **8**, 1998, 23–60.

[35] Voigtländer, J.: Concatenate, reverse and map vanish for free, *International Conference on Functional Programming, Proceedings*, pages 14–25, ACM Press, 2002.

[36] Wadler, P.: Theorems for free!, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359, ACM Press, 1989.

[37] Wadler, P., Blott, S.: How to make *ad-hoc* polymorphism less *ad hoc*, *Principles of Programming Languages, Proceedings*, pages 60–76, ACM Press, 1989.