Acta Informatica manuscript No.
(will be inserted by the editor)

Refined Typing to Localize the Impact of Forced Strictness on Free Theorems*

Daniel Seidel · Janis Voigtländer

March 25, 2011

Abstract Free theorems establish interesting properties of parametrically polymorphic functions, solely from their types, and serve as a nice proof tool. For pure and lazy functional programming languages, they can be used with very few preconditions. Unfortunately, in the presence of selective strictness, as provided in languages like Haskell, their original strength is reduced. In this paper we present an approach for overcoming this weakness in specific situations. Employing a refined type system which tracks the use of enforced strict evaluation, we rule out unnecessary restrictions that otherwise emerge. Additionally, we provide (and implement) an algorithm determining all refined types for a given term.

1 Introduction

Free theorems [26, 38] are a useful proof tool in lazy functional languages like Haskell [20], in particular for verifying program transformations [6, 7, 11, 12, 31, 33, 34], but also for other results: reduction of testing effort [2], meta-theorems about whole classes of algorithms [4, 32], solutions to the view-update problem from databases [35, 37], and reasoning about effectful programs [18, 36]. Initially, free theorems have been investigated in the pure polymorphic lambda calculus [25], additionally taking the influence of general recursion into account. But modern languages like Haskell and Clean [24] extend the pure polymorphic lambda calculus not only by a fixpoint combinator; they additionally allow forcing strictness at places selected by the programmer. Selective

* An earlier version of this paper appeared under the title "Taming Selective Strictness" in the electronic proceedings of the "4. Arbeitstagung Programmiersprachen", volume 154 of Lecture Notes in Informatics, pages 2916–2930, Gesellschaft für Informatik, 2009.

Correspondence to: Janis Voigtländer, E-mail: jv@iai.uni-bonn.de, Telephone: +49 228 734535, Fax: +49 228 734382

Daniel Seidel · Janis Voigtländer
University of Bonn
Institute for Computer Science
Römerstraße 164
53117 Bonn, Germany
E-mail: {ds,jv}@iai.uni-bonn.de

strict evaluation is in particular desirable to avoid so-called space leaks [19, Section 23.3.2]. A disadvantage of using forced strictness is the resulting weakening of *relational parametricity*, the conceptual base for free theorems.

For example, in the absence of selective strictness a free theorem establishes that *every* function $foldl :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$ satisfies the equation

$$f\ (foldl\ c\ n\ xs) = foldl\ c'\ (f\ n)\ (map\ g\ xs) \tag{1}$$

for *arbitrary choices* (appropriately typed) of c, c', n, xs, and of strict f, g ($f \perp = \perp$ and $g \perp = \perp$) such that for every x and y, $f\ (c\ x\ y) = c'\ (f\ x)\ (g\ y)$. But the Haskell standard library Data.List contains a function $foldl'$ of the mentioned type for which the equation can only be guaranteed to hold if *additionally* f is total ($f\ x \neq \perp$ for every $x \neq \perp$). The reason is a use of forced strict evaluation, via Haskell's strict evaluation primitive *seq*, inside the definition of $foldl'$ (see Section 2).

Johann and Voigtländer [13] have studied the impact of selective strictness on free theorems in detail on a global level. For the type of *foldl* above, their results indicate that in general, under the most pessimistic assumptions about uses of forced strict evaluation inside function definitions, it is also required that g is total and that $c = \perp$ iff $c' = \perp$ and for every x, $c\ x = \perp$ iff $c'\ (f\ x) = \perp$. Only then we can uphold the given free theorem. But for specific functions, like $foldl'$, fewer conditions can be sufficient. In fact, not alone *whether* selective strictness is used somewhere, but *where* it is used determines the necessity and the exact nature of restrictions. So a natural question is: How can we express detailed information about the use of *seq* such that we can go with *as few as possible* additional restrictions put on the original free theorems? That is the problem we solve in this paper: how to move from Johann and Voigtländer's all-or-nothing approach to a more localized account of the impact of selective strictness on free theorems.

Since free theorems depend only on the *type* of a term, any information used has to be part of the type signature. Hence, we track selective strictness by adding extra information in the type of a term. Thus, we will be able to determine based on the type whether a weakening of parametricity may arise. An attempt in this direction was already made when *seq* was first introduced into Haskell (version 1.3). The type class [39] Eval was introduced to make selective strictness and the resulting limitations with respect to parametricity explicit from the type. For example, $foldl'$ then had the refined type $foldl' :: \mathsf{Eval}\ \alpha \Rightarrow (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$, which would allow to conclude that f must be total (and strict, and g must be strict, and $\forall x, y.\ f\ (c\ x\ y) = c'\ (f\ x)\ (g\ y)$ must hold) in order to guarantee $f\ (foldl'\ c\ n\ xs) = foldl'\ c'\ (f\ n)\ (map\ g\ xs)$.

The controversy about originally incorporating selective strictness into Haskell, about putting the Eval type class in place to keep tabs on *seq*, and about later abandoning this mechanism again while retaining *seq*, is described by Hudak et al. [9, Section 10.3]. But even that description fails to recognize that actually the type class Eval as once present in Haskell was *not* sufficient to control adverse effects of selective strictness on parametricity. The reason is that the type class approach presumes that all necessary restrictions can be read off from constraints on type variables. And this is not actually always the case. Some restrictions arising from forcing strict evaluation *cannot* be read off in that way (see the last paragraph of Section 2 for an example). In brief, the failure of the original attempt at taming selective strictness is caused by the Haskell report version 1.3 (Section 6.2.7) mandating that "Functions as well as all other built-in types are in Eval." This predated the insights gained by Johann and Voigtländer [13] regarding

the special care that is required precisely for the interaction between selective strictness, parametricity, and function types. One way out would be to generally forbid use of *seq* on functions. But then it would not anymore be possible to write some programs that one currently *can* write in Haskell. For example, it would become impossible to use $foldl'$ in a situation where α becomes instantiated to a function type. An alternative would be to work with the type class approach, but consider function types to not in general be in Eval, instead constraining their membership more specifically by allowing type class restrictions on compound types[1]. But such an approach would lack desirable precision. Consider a function $f :: \mathsf{Eval}\ (\alpha \to \mathsf{Int}) \Rightarrow (\alpha \to \mathsf{Int}) \to (\alpha \to \mathsf{Int}) \to \mathsf{Int}$. It could be of the form $f = \lambda g\ h \to \ldots$ where *seq* is actually used only on g but not on h, or conversely. From the proposed type signature, there is no way to tell the difference.

Due to the mentioned problems, we argue that if in a future revision of the Haskell language it is decided to put selective strictness back under control, a new mechanism is needed. This paper provides such a mechanism, though in practice it would of course have to be scaled up from the calculus we study to the full language. We use a more elaborate way of tracking selective strictness than the type class approach did. Namely, we provide special annotations at the introduction of type variables but also at function types. This leads to a clear correspondence to the impact of *seq* on free theorems. Combining the insights of Johann and Voigtländer [13] with ideas of Launchbury and Paterson [15] regarding taming general recursion (where the type class approach actually *does* work to full satisfaction[2]), we present a calculus for which we provide refined free theorems via a refined type system. We then develop an algorithm computing all refined types for a given term. The algorithm has been implemented, and a web interface to it is online at `http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi`. In addition to producing the refined types, the tool also shows the corresponding (restricted) free theorems.

Formally, our system is an *annotated type system* in the terminology of [17, Section 2]. Since the annotations do not really describe intensional information about what takes place during evaluation of a program, we do not call it a type and effect system [17, Section 3]. Other ingredients we use of the methodology described in the mentioned survey article, in particular its Section 5, are *shape conformant subtyping* and *type inference*, with *constraints*. Beside the references to the literature given there, other useful background reading is [21] for type systems and associated algorithmic techniques in general, [27] for *denotational semantics*, and [3, 23] for *logical relations*. Also relevant is classical strictness analysis [10, 16], though we do not attempt to discover strict usage of arguments as resulting from "normal" execution, instead only focusing on explicitly enforced strict evaluation. Recent work by Holdermans and Hage [8] studies the interplay between these two aspects.

2 A Motivating Example

Consider the Haskell Prelude function *foldl*, its stricter variant $foldl'$ (from the Haskell standard library Data.List), and functions $foldl''$ and $foldl'''$ which force strict evaluation at rather arbitrary points, with implementations as shown in Fig. 1. Strict evaluation is enforced via *seq*, which evaluates its first argument, returns the second

[1] This is not allowed in the current language version Haskell 2010, but as an extension in the Glasgow Haskell Compiler, enabled by `-XFlexibleContexts`.

[2] We discuss their approach briefly towards the end of Section 3.

```
foldl c = fix
  (λh n ys →
       case ys of
         []     → n
         x : xs → let n' = c n x in h n' xs)

foldl' c = fix
  (λh n ys →
       case ys of
         []     → n
         x : xs → let n' = c n x in seq n' (h n' xs))

foldl'' c = fix
  (λh n ys →
       seq (c n)
         (case ys of
           []     → n
           x : xs → seq xs
             (seq x
               (let n' = c n x in h n' xs))))

foldl''' c = seq c (fix
  (λh n ys →
       case ys of
         []     → n
         x : xs → let n' = c n x in h n' xs))
```

Fig. 1 Variations of *foldl* with Different Uses of *seq*

argument if that evaluation is successful, and otherwise fails. The fixpoint combinator $fix :: (\alpha \to \alpha) \to \alpha$ captures general recursion — $fix\ g = g\ (fix\ g)$.

All four functions considered are of type $(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$, and as already mentioned the corresponding free theorem ignoring potential use of *seq* states that equation (1) from the introduction holds if f and g are strict and if for every x and y, $f\ (c\ x\ y) = c'\ (f\ x)\ (g\ y)$. Taking selective strictness into account, the situation changes and additional preconditions arise [13]. For example, for $foldl'$ the free theorem as just stated does not hold. To see this, consider equation (1) with the following instantiation:

$$\begin{array}{lll} f & = \lambda x \to \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot & g = id \\ c = c' & = \lambda x\ y \to \textbf{if } y \textbf{ then } \mathit{True} \textbf{ else } x & n = \mathit{False} \\ xs & = [\mathit{False}, \mathit{True}] & \end{array}$$

Regarding *foldl* everything is fine, but for the "strictified" $foldl'$ we get the false statement $True = \bot$. The problem here is that the use of *seq* on n' in the definition of $foldl'$ leads to an application of *seq* on $(c\ n\ False)$ in the left-hand side of equation (1) vs. an application of *seq* on $(c'\ (f\ n)\ (g\ False))$ in the corresponding right-hand side. By the condition relating f, c, c', and g, the second expression is equivalent to $f\ (c\ n\ False)$. But while for the above instantiation, $(c\ n\ False)$ is non-$\bot$, its f-image is $\bot$. This results in the harmful difference between the impact of *seq* on the left- and right-hand sides of equation (1) for $foldl'$ with the above instantiation. To recover equivalence here, it suffices to restrict f to be total. In fact, for $foldl'$ *every* instantiation in which f is total, in addition to the conditions given above, will make equation (1) true. But if we regard the functions $foldl''$ and $foldl'''$, for which the instantiation given above actually does *not* break the original free theorem, we will encounter the necessity of further restrictions. Specifically, consider each of the following instantiations:

$$\begin{array}{llllll} f = id & g = t_1 & c = t_2 & c' = t_2 & n = \mathit{True} & xs = [\mathit{False}] \\ f = id & g = id & c = t_3 & c' = t_4 & n = \mathit{False} & xs = [\,] \\ f = id & g = id & c = \bot & c' = \lambda x \to \bot & n = \mathit{False} & xs = [\,] \end{array}$$

where $t_1 = \lambda x \to \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$, $t_2 = \lambda x\ y \to \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } y$, $t_3 = \lambda x\ y \to \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$, and $t_4 = \lambda x \to \textbf{if } x \textbf{ then } \lambda y \to \mathit{True} \textbf{ else } \bot$. For each of these instantiations, equation (1) holds for *foldl* and $foldl'$, but the first

$\tau ::= \alpha \mid [\tau] \mid \tau \to \tau$

$t ::= x \mid []_\tau \mid t : t \mid \mathbf{case}\ t\ \mathbf{of}\ \{[] \to t;\ x : x \to t\} \mid \lambda x :: \tau.t \mid t\ t \mid \mathbf{fix}\ t \mid \mathbf{let!}\ x = t\ \mathbf{in}\ t$

Fig. 2 Syntax of Types τ and Terms t

and the second instantiation break the equation for *foldl''*, while the last instantiation breaks the equation for *foldl'''*. These three failures are caused by different uses of *seq*, which enforce different restrictions if we want to regain equation (1). Only the use of *seq* on the list *xs* causes no additional restriction.

We already mentioned in the introduction that to guarantee equation (1) for *all* functions of *foldl*'s type (including *foldl'*, *foldl''*, and *foldl'''*), we need to restrict both f and g to be total and additionally need to require $c = \bot$ iff $c' = \bot$ and for every x, $c\ x = \bot$ iff $c'\ (f\ x) = \bot$, in addition to the conditions from the original free theorem. Our aim is to instead tailor the set of restrictions needed to the actual kind of use that is made of selective strictness in a given function. Indeed, our resulting tool (`http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi`) will help to clarify which *seq* in Fig. 1 causes which restriction (see Section 6). To reiterate that the Eval type class mechanism of Haskell version 1.3 was not sound in terms of detecting all necessary restrictions, note that *foldl'''* would have incurred no Eval-constraint in its type at all, hence no need for additional restrictions would have been discovered, but as seen above, the use of *seq* on *c does* cause problems.

3 The Calculus and Standard Parametricity

We set out from a standard denotational semantics for a polymorphic lambda calculus, called **PolySeq**, that corresponds to a small core of Haskell.

The syntax of types and terms is given in Fig. 2, where α ranges over type variables and x ranges over term variables. We include lists as representative for algebraic data types. General recursion is captured via a fixpoint primitive, while selective strictness (à la *seq*) is provided via a strict-let construct as also found in the functional language Clean. Note that there is no case $\forall\alpha.\tau$ in the type grammar, and no type abstraction and application formers in the term grammar. In fact, our calculus is simply typed but with type variables. The technical report version [28] considers true polymorphism, including higher-rank polymorphism, but here we simplified for the sake of presentation. All the interesting points we want to make about our approach can still be made.

Figs. 3 and 4 give the typing axioms and rules for the calculus. Standard conventions apply here. In particular, typing environments Γ are unordered sets of the form $\alpha_1, \ldots, \alpha_k, x_1 :: \tau_1, \ldots, x_l :: \tau_l$ with distinct α_i and x_j, and in a typing judgement $\Gamma \vdash t :: \tau$ all variables occurring in a τ_j or in τ have to be among the listed $\alpha_1, \ldots, \alpha_k$.

For example, the standard Haskell function *map* can be defined as the following term and then satisfies $\alpha, \beta \vdash map :: \tau$, where $\tau = (\alpha \to \beta) \to [\alpha] \to [\beta]$:

$$\mathbf{fix}\ (\lambda m :: \tau.\lambda h :: \alpha \to \beta.\lambda l :: [\alpha].\mathbf{case}\ l\ \mathbf{of}\ \{[] \to []_\beta;\ x : y \to (h\ x) : (m\ h\ y)\})$$

The denotational semantics interprets types as *pointed complete partial orders* (for short, *pcpos*; least element always denoted $\bot$). The definition in Fig. 5, assuming θ to be a mapping from type variables to pcpos, is entirely standard. The operation $lift_\bot$ takes a complete partial order, adds a new element $\bot$ to the carrier set, defines this new $\bot$ to be below every other element, and leaves the ordering otherwise unchanged.

$$\Gamma, x :: \tau \vdash x :: \tau \ (\textsc{Var}) \qquad \Gamma \vdash [\,]_\tau :: [\tau] \ (\textsc{Nil})$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \ (\textsc{Cons})$$

$$\frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau_2 \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1;\ x_1 : x_2 \to t_2\}) :: \tau_2} \ (\textsc{Case})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: (\tau_1 \to \tau_2)} \ (\textsc{Abs})$$

$$\frac{\Gamma \vdash t_1 :: (\tau_1 \to \tau_2) \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2} \ (\textsc{App}) \qquad \frac{\Gamma \vdash t :: (\tau \to \tau)}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau} \ (\textsc{Fix})$$

Fig. 3 Some Typing Axioms and Rules in **PolySeq** (and Later **PolySeq***)

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2} \ (\textsc{SLet})$$

Fig. 4 An Additional Typing Rule in **PolySeq**

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![[\tau]]\!]_\theta &= \mathit{gfp}\,(\lambda S.\mathit{lift}_\perp\,(\{[\,]\} \cup \{(a : b) \mid a \in [\![\tau]\!]_\theta,\, b \in S\})) \\
[\![\tau_1 \to \tau_2]\!]_\theta &= \mathit{lift}_\perp\,\{f : [\![\tau_1]\!]_\theta \to [\![\tau_2]\!]_\theta\}
\end{aligned}$$

Fig. 5 Semantics of Types

To avoid confusion, the original elements are tagged, i.e., $\mathit{lift}_\perp\, S = \{\perp\} \cup \{\lfloor s \rfloor \mid s \in S\}$. For list types, prior to lifting, $[\,]$ is only related to itself, while the ordering between "$(- : -)$"-values is component-wise. Also note the use of the *greatest* fixpoint, under set inclusion, to capture infinite lists. The function space lifted in the definition of $[\![\tau_1 \to \tau_2]\!]_\theta$ is the one of monotonic and continuous maps between $[\![\tau_1]\!]_\theta$ and $[\![\tau_2]\!]_\theta$, ordered point-wise.

The semantics of terms, given in Fig. 6, is also standard. It uses λ for denoting anonymous functions, and the following operator:

$$h\ \$\ a = \begin{cases} f\ a & \text{if } h = \lfloor f \rfloor \\ \perp & \text{if } h = \perp \end{cases}$$

The expression $\bigsqcup_{n \geq 0}\ ([\![t]\!]_\eta\ \$)^n\ \perp$ in the definition for **fix** means the supremum of the chain $\perp \sqsubseteq ([\![t]\!]_\eta\ \$\ \perp) \sqsubseteq ([\![t]\!]_\eta\ \$\ ([\![t]\!]_\eta\ \$\ \perp)) \cdots$. Altogether, we have that if $\Gamma \vdash t :: \tau$ and $\eta(x) \in [\![\tau']\!]_\theta$ for every $x :: \tau'$ occurring in Γ, then $[\![t]\!]_\eta \in [\![\tau]\!]_\theta$.

The key to parametricity results is the definition of a family of relations by induction on a calculus' type structure. The appropriate such *logical relation* for our current setting is defined in Fig. 7, assuming ρ to be a mapping from type variables to binary relations between pcpos. The operation *list* takes a relation $\mathcal{R}$ and maps it to

$$\mathit{list}\,\mathcal{R} = \mathit{gfp}\,(\lambda\mathcal{S}.\{(\perp, \perp),\, (\lfloor [\,] \rfloor, \lfloor [\,] \rfloor)\} \cup \{(\lfloor a : b \rfloor, \lfloor c : d \rfloor) \mid (a, c) \in \mathcal{R},\, (b, d) \in \mathcal{S}\})$$

where again the greatest fixpoint is taken.

For two pcpos D_1 and D_2, let $\mathit{Rel}(D_1, D_2)$ collect all relations between them that are *strict*, *continuous*, and *bottom-reflecting*. Strictness and continuity are just the standard notions, i.e., membership of the pair $(\perp, \perp)$ and closure under suprema. A relation

$$
\begin{array}{ll}
[\![x]\!]_\eta & = \eta(x) \\
[\![[\,]_\tau]\!]_\eta & = \lfloor[\,]\rfloor \\
[\![t_1 : t_2]\!]_\eta & = \lfloor [\![t_1]\!]_\eta : [\![t_2]\!]_\eta \rfloor \\
[\![\textbf{case } t \textbf{ of } \{[\,] \to t_1;\ x_1 : x_2 \to t_2\}]\!]_\eta & = \begin{cases} [\![t_1]\!]_\eta & \text{if } [\![t]\!]_\eta = \lfloor[\,]\rfloor \\ [\![t_2]\!]_{\eta[x_1 \mapsto a,\, x_2 \mapsto b]} & \text{if } [\![t]\!]_\eta = \lfloor a : b \rfloor \\ \bot & \text{if } [\![t]\!]_\eta = \bot \end{cases} \\
[\![\lambda x :: \tau . t]\!]_\eta & = \lfloor \lambda a . [\![t]\!]_{\eta[x \mapsto a]} \rfloor \\
[\![t_1\ t_2]\!]_\eta & = [\![t_1]\!]_\eta\ \$\ [\![t_2]\!]_\eta \\
[\![\textbf{fix } t]\!]_\eta & = \bigsqcup_{n \geq 0}\ (([\![t]\!]_\eta\ \$)^n\ \bot \\
[\![\textbf{let! } x = t_1 \textbf{ in } t_2]\!]_\eta & = \begin{cases} [\![t_2]\!]_{\eta[x \mapsto a]} & \text{if } [\![t_1]\!]_\eta = a \neq \bot \\ \bot & \text{if } [\![t_1]\!]_\eta = \bot \end{cases}
\end{array}
$$

Fig. 6 Semantics of Terms

$$
\begin{array}{ll}
\Delta_{\alpha,\rho} & = \rho(\alpha) \\
\Delta_{[\tau],\rho} & = \mathit{list}\ \Delta_{\tau,\rho} \\
\Delta_{\tau_1 \to \tau_2,\rho} & = \{(f,g) \mid f = \bot \text{ iff } g = \bot,\ \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau_2,\rho}\}
\end{array}
$$

Fig. 7 Standard Logical Relation

$\mathcal{R}$ is bottom-reflecting if $(a,b) \in \mathcal{R}$ implies that $a = \bot$ iff $b = \bot$. The corresponding explicit condition on f and g in the definition of $\Delta_{\tau_1 \to \tau_2,\rho}$ in Fig. 7 serves the purpose of ensuring that bottom-reflection is preserved throughout the logical relation. Overall, induction on τ gives the following important lemma, where *Rel* is the union of all $\mathit{Rel}(D_1, D_2)$.

Lemma 1 *If ρ maps to relations in Rel, then $\Delta_{\tau,\rho} \in \mathit{Rel}$.*

The lemma is crucial for then proving the following theorem.

Theorem 1 (Parametricity, PolySeq) *If $\Gamma \vdash t :: \tau$, then for every θ_1, θ_2, ρ, η_1, and η_2 such that*

- *for every α occurring in Γ, $\rho(\alpha) \in \mathit{Rel}(\theta_1(\alpha), \theta_2(\alpha))$ and*
- *for every $x :: \tau'$ occurring in Γ, $(\eta_1(x), \eta_2(x)) \in \Delta_{\tau',\rho}$,*

we have $([\![t]\!]_{\eta_1}, [\![t]\!]_{\eta_2}) \in \Delta_{\tau,\rho}$.

Proof The proof follows [13] and is by induction over typing derivations. We only show a few representative induction cases here. For (Abs) we have

$$
\begin{array}{ll}
& ([\![\lambda x :: \tau_1 . t]\!]_{\eta_1}, [\![\lambda x :: \tau_1 . t]\!]_{\eta_2}) \in \Delta_{\tau_1 \to \tau_2,\rho} \\
\Leftrightarrow & (\lfloor \lambda a . [\![t]\!]_{\eta_1[x \mapsto a]} \rfloor, \lfloor \lambda b . [\![t]\!]_{\eta_2[x \mapsto b]} \rfloor) \in \Delta_{\tau_1 \to \tau_2,\rho} \\
\Leftrightarrow & \forall (a,b) \in \Delta_{\tau_1,\rho}.\ ([\![t]\!]_{\eta_1[x \mapsto a]}, [\![t]\!]_{\eta_2[x \mapsto b]}) \in \Delta_{\tau_2,\rho}
\end{array}
$$

so the induction hypothesis suffices. For (Fix) we have

$$
\begin{array}{ll}
& ([\![\textbf{fix } t]\!]_{\eta_1}, [\![\textbf{fix } t]\!]_{\eta_2}) \in \Delta_{\tau,\rho} \\
\Leftrightarrow & (\bigsqcup_{n \geq 0}\ (([\![t]\!]_{\eta_1}\ \$)^n\ \bot, \bigsqcup_{n \geq 0}\ (([\![t]\!]_{\eta_2}\ \$)^n\ \bot) \in \Delta_{\tau,\rho} \\
\Leftarrow & \forall n \geq 0.\ ((([\![t]\!]_{\eta_1}\ \$)^n\ \bot, (([\![t]\!]_{\eta_2}\ \$)^n\ \bot) \in \Delta_{\tau,\rho} \\
\Leftarrow & \forall (a,b) \in \Delta_{\tau,\rho}.\ ([\![t]\!]_{\eta_1}\ \$\ a, [\![t]\!]_{\eta_2}\ \$\ b) \in \Delta_{\tau,\rho} \\
\Leftarrow & ([\![t]\!]_{\eta_1}, [\![t]\!]_{\eta_2}) \in \Delta_{\tau \to \tau,\rho}
\end{array}
$$

and therefore the induction hypothesis suffices again. Note that we use the continuity of $\Delta_{\tau,\rho}$ in the first implication and the strictness of $\Delta_{\tau,\rho}$ in the second implication

here, both given by Lemma 1. For (SLET) we have to show that the values

$$\begin{cases} [\![t_2]\!]_{\eta_1[x\mapsto a]} & \text{if } [\![t_1]\!]_{\eta_1} = a \neq \bot \\ \bot & \text{if } [\![t_1]\!]_{\eta_1} = \bot \end{cases}$$

and

$$\begin{cases} [\![t_2]\!]_{\eta_2[x\mapsto b]} & \text{if } [\![t_1]\!]_{\eta_2} = b \neq \bot \\ \bot & \text{if } [\![t_1]\!]_{\eta_2} = \bot \end{cases}$$

are related by $\Delta_{\tau_2,\rho}$ if:

- $([\![t_1]\!]_{\eta_1}, [\![t_1]\!]_{\eta_2}) \in \Delta_{\tau_1,\rho}$ and
- for every $(a,b) \in \Delta_{\tau_1,\rho}$, $([\![t_2]\!]_{\eta_1[x\mapsto a]}, [\![t_2]\!]_{\eta_2[x\mapsto b]}) \in \Delta_{\tau_2,\rho}$.

By bottom-reflection of $\Delta_{\tau_1,\rho}$, which holds due to Lemma 1, it suffices to consider the following two cases:

1. $[\![t_1]\!]_{\eta_1} = a \neq \bot$ and $[\![t_1]\!]_{\eta_2} = b \neq \bot$, in which case we are done by the known $([\![t_2]\!]_{\eta_1[x\mapsto a]}, [\![t_2]\!]_{\eta_2[x\mapsto b]}) \in \Delta_{\tau_2,\rho}$ for every $(a,b) \in \Delta_{\tau_1,\rho}$,
2. $[\![t_1]\!]_{\eta_1} = \bot$ and $[\![t_1]\!]_{\eta_2} = \bot$, in which case we are done by $(\bot,\bot) \in \Delta_{\tau_2,\rho}$, which holds by the strictness of $\Delta_{\tau_2,\rho}$ (cf. Lemma 1 again).

The remaining cases are detailed in Appendix A of our technical report [28]. □

If we did not have **let!** in the calculus, then instead of requiring strictness, continuity, and bottom-reflection of all relations, the first two would have been enough (and the condition "$f = \bot$ iff $g = \bot$" in the definition of $\Delta_{\tau_1\to\tau_2,\rho}$ could have been dropped). That would have led to the version of equation (1) from the introduction where f and g must be strict but not necessarily total (and where $c = \bot$ iff $c' = \bot$ and $c\ x = \bot$ iff $c'\ (f\ x) = \bot$ for every x are not required). As visible from the above proof, strictness and continuity are already required when **fix** is in the calculus. Launchbury and Paterson [15] proposed a refined type system that keeps track of uses of **fix** and admits a refined notion of parametricity in which as few as possible of these conditions are imposed, depending on the recorded information. Specifically, they introduce a type class Pointed, where type variables must be explicitly constrained if they are to be considered to be in that type class, where list types are always in Pointed, and where a function type is in Pointed if the result type is:

$$\frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma \vdash (\tau_1 \to \tau_2) \in \mathsf{Pointed}}$$

Then, though expressed with different notation, they revise the typing rule (FIX) to

$$\frac{\Gamma \vdash \tau \in \mathsf{Pointed} \qquad \Gamma \vdash t :: (\tau \to \tau)}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau}$$

and similarly add $\Gamma \vdash \tau_2 \in \mathsf{Pointed}$ as premise to typing rule (CASE). This allows them to prove a version of Theorem 1 (for the definition of Δ without "$f = \bot$ iff $g = \bot$" in $\Delta_{\tau_1\to\tau_2,\rho}$) in which the $\rho(\alpha)$ need not be bottom-reflecting and need to be strict only if α is in Pointed. Our aim is to succeed similarly for **let!** and bottom-reflection.

We could have tried to simultaneously keep track of **fix** and **let!**, and thus get very fine-grained results about where any of strictness, totality/bottom-reflection, and "$f = \bot$ iff $g = \bot$"-conditions are needed. For simplicity we do not do so, instead focusing on only **let!** here. That is, we do not introduce Pointed and we keep the original versions of typing rules (FIX) and (CASE) from Fig. 3.

4 Refining the Calculus and the Parametricity Theorem

If we recall the fold functions from Fig. 1 and the "*seq*-ignoring" version of the corresponding free theorem, stated in equation (1) in the introduction, we can compare that version with the "*seq*-safe" version arising from Theorem 1. The safe theorem requires f and g to be total, $c = \bot$ iff $c' = \bot$, and for every x, $c\ x = \bot$ iff $c'\ (f\ x) = \bot$, in addition to the restrictions from the less safe theorem. Under these combined conditions, it delivers equation (1) and additionally that $\mathit{foldl}\ c = \bot$ iff $\mathit{foldl}\ c' = \bot$, as well as that for every n, $\mathit{foldl}\ c\ n = \bot$ iff $\mathit{foldl}\ c'\ (f\ n) = \bot$. All this is obtained by invoking Theorem 1 as $(\llbracket \mathit{foldl} \rrbracket_\emptyset, \llbracket \mathit{foldl} \rrbracket_\emptyset) \in \Delta_{(\alpha\to\beta\to\alpha)\to\alpha\to[\beta]\to\alpha,\rho}$, unfolding a number of definitions, setting $\rho(\alpha) = \{(x_1, x_2) \mid \llbracket f \rrbracket_\emptyset\ \$\ x_1 = x_2\}$ and $\rho(\beta) = \{(y_1, y_2) \mid \llbracket g \rrbracket_\emptyset\ \$\ y_1 = y_2\}$ for some f and g, and using the straightforward relationship that then $\mathit{list}\,\rho(\beta) = \{(xs_1, xs_2) \mid (\llbracket \mathit{map} \rrbracket_\emptyset\ \$\ \llbracket g \rrbracket_\emptyset)\ \$\ xs_1 = xs_2\}$. The strictness and totality conditions on f and g stem from the requirement that $\rho(\alpha), \rho(\beta) \in \mathit{Rel}$ and Rel contains only relations that are strict and bottom-reflecting.[3]

As already motivated, the additional restrictions of the "*seq*-safe" over the "*seq*-ignoring" version arise from different potential uses of forced strictness. That is, each restriction is only necessary if enforced strict evaluation is used in some special way. Hence, it is reasonable to make selective strictness (and the "places" of its use) visible from the type of a term. In particular, the use of enforced strictness on elements of a type should be visible for type variables and function types. Forcing evaluation on lists is nothing to worry about, because it anyway can be simulated by a case statement. Thus, we want to distinguish function types and type variables on whose elements *seq*/**let**! is used from those on whose elements it is not so. More precisely, we want to distinguish function types and type variables on whose elements forcing evaluation is *allowed* from those on whose elements it is *not allowed*. Therefore we introduce annotations ε and $\circ$ at occurrences of the type constructor $\to$ as well as at type variables in the typing environment. An annotation ε signifies that forcing evaluation is allowed on the entity in question, whereas an annotation $\circ$ prevents the use of selective strictness at a certain place. Recalling the example foldl'' from Section 2, one of its refined types would be $(\alpha \to^\circ \beta \to^\varepsilon \alpha) \to^\varepsilon \alpha \to^\varepsilon [\beta] \to^\varepsilon \alpha$ in typing environment $\alpha^\circ, \beta^\varepsilon$. That is actually as good as it gets, because:

- In general, for getting stronger free theorems, it is preferable to have as many type variables as possible $\circ$-annotated and to whenever possible use ε on function arrows in positive positions and $\circ$ on function arrows in negative positions.
- For foldl'', we cannot have a $\circ$-annotation on β, because of the *seq* on x.
- For foldl'', we cannot have a $\circ$-annotation on the second arrow in the function argument type $(\alpha \to \beta \to \alpha)$, because of the *seq* on $(c\ n)$.

For convenience, in the remainder of the paper we take ε to be the invisible annotation and almost always drop it.[4]

Using the axiom system from Fig. 8 we define exactly the types on whose elements we allow the use of selective strictness, by collecting them in the class Seqable. Having now an explicit way to describe which types support selective strict evaluation, we can

[3] Recall that in the introduction we said a function f is strict if $f\ \bot = \bot$ and is total if $f\ x \neq \bot$ for every $x \neq \bot$. Here, more formally, we use corresponding conditions involving the $\$$-operator.

[4] In fact, being able to do this (ε = empty) was the only motivation for choosing that symbol. No specific motivation exists for choosing $\circ$, but we have to use *some* symbol.

$$\Gamma \vdash [\tau] \in \mathsf{Seqable}\ (\text{C-List})$$

$$\Gamma \vdash (\tau_1 \to^{\varepsilon} \tau_2) \in \mathsf{Seqable}\ (\text{C-Arrow}) \qquad \alpha^{\varepsilon}, \Gamma \vdash \alpha \in \mathsf{Seqable}\ (\text{C-Var})$$

Fig. 8 Class Membership Axioms for Seqable in **PolySeq*** (and Later **PolySeq**$^{+}$)

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\text{SLet'})$$

Fig. 9 Replacement for (SLet) from Fig. 4 in **PolySeq***

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: (\tau_1 \to^{\circ} \tau_2)}\ (\text{Abs}_{\circ}) \qquad \frac{\Gamma \vdash t_1 :: (\tau_1 \to^{\circ} \tau_2) \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\text{App}_{\circ})$$

$$\frac{\Gamma \vdash t :: (\tau \to^{\circ} \tau)}{\Gamma \vdash (\mathbf{fix}\ t) :: \tau}\ (\text{Fix}_{\circ}) \qquad \frac{\Gamma \vdash t :: \tau_1 \qquad \tau_1 \preceq \tau_2}{\Gamma \vdash t :: \tau_2}\ (\text{Sub})$$

Fig. 10 New Typing Rules in **PolySeq*** (Some are Variants of Rules from Fig. 3)

restrict the typing rule (SLet) to these types. The new rule is given in Fig. 9. The other typing axioms and rules of **PolySeq**, as shown in Fig. 3, remain unchanged, but we add $\circ$-annotated versions (Abs$_{\circ}$), (App$_{\circ}$), and (Fix$_{\circ}$), with all explicit occurrences of $\to$ annotated by $\circ$. These additional rules are shown in Fig. 10, along with a last rule required for the extended calculus, (Sub), which we explain next.

The motivation for (Sub) is that *allowing* the use of selective strictness does not entail *insisting* on it. Consider the types $(\tau_1 \to^{\circ} \tau_2) \to [\tau_3]$ and $(\tau_1 \to \tau_2) \to [\tau_3]$. All terms typable to the first one should be typable to the second one as well. After all, the first type promises that we have a function producing a list of type $[\tau_3]$ from a function mapping τ_1 to τ_2, and that we know that inside the function this functional argument is not forcedly evaluated (other than by possibly applying it to an argument of type τ_1, of course). Clearly, such a function is also a function producing a list of type $[\tau_3]$ from a function mapping τ_1 to τ_2 while *being allowed to* use forced evaluation on the functional argument. It then simply makes no use of that "being allowed to". On the other hand, not every function of type $(\tau_1 \to \tau_2) \to [\tau_3]$ should be considered to be of type $(\tau_1 \to^{\circ} \tau_2) \to [\tau_3]$ as well. For example, the **PolySeq**-term $\lambda f :: \tau_1 \to \tau_2.\mathbf{let!}\ x = f\ \mathbf{in}\ []_{\tau_3}$ should only be typable to $(\tau_1 \to \tau_2) \to [\tau_3]$ but not to $(\tau_1 \to^{\circ} \tau_2) \to [\tau_3]$. (And annotating it to get $\lambda f :: \tau_1 \to^{\circ} \tau_2.\mathbf{let!}\ x = f\ \mathbf{in}\ []_{\tau_3}$ should clearly make it not typable at all.) All this is guaranteed by rule (Sub) in connection with the subtype relation defined in Fig. 11. In the parameterized rule family $(\text{S-Arrow}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu' \leqslant \nu}$, and generally in what follows, we take $\{\circ, \varepsilon\}$ to be the ordered set of annotations with $\circ < \varepsilon$. As a consequence, the rule family represents three rules (S-Arrow$_{\varepsilon,\varepsilon}$), (S-Arrow$_{\circ,\circ}$), and (S-Arrow$_{\varepsilon,\circ}$), while there is no corresponding rule (S-Arrow$_{\circ,\varepsilon}$). Thus, the subtyping system ensures that a Seqable supertype has only Seqable subtypes. We can think of this as follows: the set of functions on which we do allow use of selective strictness is a subset of the set of all functions. Together with the standard contravariant interpretation of subtyping at function argument types — $\tau_1 \preceq \sigma_1$ in $(\text{S-Arrow}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu' \leqslant \nu}$ — we get exactly the desired behavior, in the above example and in general.

The axiom and rule systems just described (i.e., Fig. 3 plus Figs. 8–11) set up a new calculus **PolySeq***. To take over the term and type semantics from **PolySeq**, we define an *annotation eraser* $|\cdot|$, removing all $\circ$-annotations when applied to a term,

$$\alpha \preceq \alpha \text{ (S-VAR)} \qquad \frac{\tau \preceq \tau'}{[\tau] \preceq [\tau']} \text{ (S-LIST)}$$

$$\frac{\tau_1 \preceq \sigma_1 \qquad \sigma_2 \preceq \tau_2}{(\sigma_1 \to^{\nu} \sigma_2) \preceq (\tau_1 \to^{\nu'} \tau_2)} \text{ (S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu' \leqslant \nu}$$

Fig. 11 Subtyping Axiom and Rules in **PolySeq*** (and later **PolySeq**$^+$)

type, or typing environment. It also allows us to establish the sets of typable terms in **PolySeq*** and in **PolySeq** to be equivalent in the sense of the following observation.

Observation 1 *If Γ, t, and τ are such that $\Gamma \vdash t :: \tau$ holds in **PolySeq**, then $\Gamma \vdash t :: \tau$ holds in **PolySeq***. Conversely, if Γ, t, and τ are such that $\Gamma \vdash t :: \tau$ holds in **PolySeq***, then $|\Gamma| \vdash |t| :: |\tau|$ holds in **PolySeq**.*

The point of restricting use of selective strictness to terms whose types are in Seqable was to allow the relational interpretation of all other types to be non-bottom-reflecting and thus to get rid of restrictions on free theorems derived from Theorem 1. Hence, our refined parametricity theorem will not require relations $\rho(\alpha)$ for $^\circ$-annotated α to be bottom-reflecting. Moreover, we now allow the relational action for $\to^\circ$ to forget about bottom-reflection, and define it as follows:

$$\Delta_{\tau_1 \to^\circ \tau_2, \rho} = \{(f, g) \mid \forall (a, b) \in \Delta_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau_2,\rho}\}$$

The other relational actions remain as in **PolySeq** (cf. Fig. 7).

Before we give the refined parametricity theorem for **PolySeq***, we have to establish that the logical relation just defined is strict and continuous for all types and additionally bottom-reflecting for all types in Seqable, even when assuming bottom-reflection only for relations interpreting type variables that are ε-annotated in the typing environment. Also, the impact of subtyping on the logical relation has to be clarified. It turns out that a subtype relationship between two types has a very natural interpretation as the relation corresponding to the subtype being a subset of the relation corresponding to the supertype. The two lemmas to follow next establish these facts.

For two pcpos D_1 and D_2, let $Rel^\circ(D_1, D_2)$ collect all relations between them that are strict and continuous, *but not necessarily bottom-reflecting*. Also, let Rel° be the union of all $Rel^\circ(D_1, D_2)$. Note that Rel is properly contained in Rel°.

Lemma 2

1. *If ρ maps to relations in Rel°, then $\Delta_{\tau,\rho} \in Rel^\circ$.*
2. *If $\Gamma \vdash \tau \in$ Seqable, then for every ρ such that*
   - *for every α° occurring in Γ, $\rho(\alpha) \in Rel^\circ$, and*
   - *for every α^ε occurring in Γ, $\rho(\alpha) \in Rel$,*

   we have $\Delta_{\tau,\rho} \in Rel$.

Proof By induction on τ and case distinction on $\Gamma \vdash \tau \in$ Seqable. □

Lemma 3 *If $\tau_1 \preceq \tau_2$, then $\Delta_{\tau_1,\rho} \subseteq \Delta_{\tau_2,\rho}$.*

Proof By induction on derivation trees built from the axiom and rules from Fig. 11, the only really interesting case being (S-ARROW$_{\varepsilon,\circ}$), where we use that always $\Delta_{\tau \to \tau',\rho} \subseteq \Delta_{\tau \to^\circ \tau',\rho}$.[5] □

[5] ... in contrast to $\Delta_{\tau \to^\circ \tau',\rho} \subseteq \Delta_{\tau \to \tau',\rho}$, which does not in general hold, justifying why there is no rule (S-ARROW$_{\circ,\varepsilon}$).

Now we can state and prove a refined parametricity theorem for **PolySeq*** that gives stronger free theorems if we localize the use of selective strictness.

Theorem 2 (Parametricity, PolySeq*) *If $\Gamma \vdash t :: \tau$ in **PolySeq***, then for every θ_1, θ_2, ρ, η_1, and η_2 such that*

- *for every α° occurring in Γ, $\rho(\alpha) \in Rel^\circ(\theta_1(\alpha), \theta_2(\alpha))$,*
- *for every α^ε occurring in Γ, $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$, and*
- *for every $x :: \tau'$ occurring in Γ, $(\eta_1(x), \eta_2(x)) \in \Delta_{\tau',\rho}$,*

we have $(\llbracket |t| \rrbracket_{\eta_1}, \llbracket |t| \rrbracket_{\eta_2}) \in \Delta_{\tau,\rho}$.

Proof The proof is very similar to the one of Theorem 1. The only two really interesting induction cases are those for (Sub) and (SLet'). The former is simply by Lemma 3. For (SLet') we now have to show that the values

$$\begin{cases} \llbracket |t_2| \rrbracket_{\eta_1[x \mapsto a]} & \text{if } \llbracket |t_1| \rrbracket_{\eta_1} = a \neq \bot \\ \bot & \text{if } \llbracket |t_1| \rrbracket_{\eta_1} = \bot \end{cases}$$

and

$$\begin{cases} \llbracket |t_2| \rrbracket_{\eta_2[x \mapsto b]} & \text{if } \llbracket |t_1| \rrbracket_{\eta_2} = b \neq \bot \\ \bot & \text{if } \llbracket |t_1| \rrbracket_{\eta_2} = \bot \end{cases}$$

are related by $\Delta_{\tau_2,\rho}$ if:

- $\Gamma \vdash \tau_1 \in \mathsf{Seqable}$,
- $(\llbracket |t_1| \rrbracket_{\eta_1}, \llbracket |t_1| \rrbracket_{\eta_2}) \in \Delta_{\tau_1,\rho}$, and
- for every $(a, b) \in \Delta_{\tau_1,\rho}$, $(\llbracket |t_2| \rrbracket_{\eta_1[x \mapsto a]}, \llbracket |t_2| \rrbracket_{\eta_2[x \mapsto b]}) \in \Delta_{\tau_2,\rho}$.

By bottom-reflection of $\Delta_{\tau_1,\rho}$, which holds due to Lemma 2(2), it suffices to consider the following two cases:

1. $\llbracket |t_1| \rrbracket_{\eta_1} = a \neq \bot$ and $\llbracket |t_1| \rrbracket_{\eta_2} = b \neq \bot$, in which case we are done by the known $(\llbracket |t_2| \rrbracket_{\eta_1[x \mapsto a]}, \llbracket |t_2| \rrbracket_{\eta_2[x \mapsto b]}) \in \Delta_{\tau_2,\rho}$ for $(a, b) \in \Delta_{\tau_1,\rho}$,
2. $\llbracket |t_1| \rrbracket_{\eta_1} = \bot$ and $\llbracket |t_1| \rrbracket_{\eta_2} = \bot$, in which case we are done by $(\bot, \bot) \in \Delta_{\tau_2,\rho}$, which holds by the strictness of $\Delta_{\tau_2,\rho}$ (cf. Lemma 2(1)).

□

We end this section with an example of a refined typing and the corresponding free theorem. Recall the function *foldl''* from Section 2 once again. As already mentioned, it can be typed in **PolySeq*** to $(\alpha \to^\circ \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$ in typing environment α°, β. Actually, more precisely, the annotated term

$$\begin{aligned} t = \lambda c :: (\alpha \to^\circ \beta \to \alpha).\mathbf{fix}\ (\lambda h :: (\alpha \to [\beta] \to \alpha).\lambda n :: \alpha.\lambda ys :: [\beta]. \\ \mathbf{let!}\ z = c\ n\ \mathbf{in} \\ \mathbf{case}\ ys\ \mathbf{of}\ \{[\,] \to n; \\ x : xs \to \mathbf{let!}\ xs' = xs\ \mathbf{in} \\ \mathbf{let!}\ x' = x\ \mathbf{in} \\ h\ (c\ n\ x')\ xs'\}) \end{aligned}$$

can be typed so. We have $|t| = \mathit{foldl''}$ — or, since we have not explicitly given a definition for *foldl''* in **PolySeq** syntax, at least $\llbracket |t| \rrbracket_\eta = \llbracket \mathit{foldl''} \rrbracket_\eta$ for every η. Hence, we can apply Theorem 2 to establish conditions under which equation (1) holds for *foldl''*. What we get is that equation (1) holds for *foldl''* even if f is not total and

$c = \bot$ iff $c' = \bot$ does not hold, in contrast to what was required by the safe free theorem for the completely unannotated type as stated at the beginning of the current section. That g still needs to be total is caused by the fact that β is not $\circ$-annotated and thus $\rho(\beta) = \{(y_1, y_2) \mid [\![g]\!]_\emptyset \ \$ \ y_1 = y_2\} \in \mathit{Rel}$ still needs to be bottom-reflecting, in contrast to $\rho(\alpha) = \{(x_1, x_2) \mid [\![f]\!]_\emptyset \ \$ \ x_1 = x_2\} \in \mathit{Rel}^\circ$ for the $\circ$-annotated α. That we still need $c\ x = \bot$ iff $c'\ (f\ x) = \bot$ for every x is caused by the second arrow in the function argument type $(\alpha \to^\circ \beta \to \alpha)$ not being $\circ$-annotated, so that at some point in the derivation of the free theorem we encounter the precondition that for every $(x_1, x_2) \in \Delta_{\alpha,\rho}$, $([\![c]\!]_\emptyset \ \$ \ x_1, [\![c']\!]_\emptyset \ \$ \ x_2) \in \Delta_{\beta\to\alpha,\rho} = \{(p, q) \mid p = \bot \text{ iff } q = \bot, \forall (a, b) \in \Delta_{\beta,\rho}.\ \cdots\}$. In contrast, the condition $c = \bot$ iff $c' = \bot$ disappears since the first arrow in the mentioned function argument type *is* $\circ$-annotated, so that the corresponding condition is "only" $([\![c]\!]_\emptyset, [\![c']\!]_\emptyset) \in \Delta_{\alpha\to^\circ(\beta\to\alpha),\rho} = \{(p, q) \mid \forall (a, b) \in \Delta_{\alpha,\rho}.\ \cdots\}$ without "$p = \bot$ iff $q = \bot$".

5 Effectively Obtaining all Permissible Types

The calculus **PolySeq*** enables refined typing for all terms typable in **PolySeq**. Our final aim is to provide automatic type refinement for given terms with standard (i.e., **PolySeq**) typings. Hence, the intended use of **PolySeq*** will be to input a **PolySeq** term t and to find all permissible types that t, for some concrete setting of annotations in its syntactic type components (e.g., at occurrences of the empty list or in λ-abstractions), is typable to in **PolySeq***.

Unfortunately, **PolySeq*** is not suitable for an algorithmic use in its current form. The rule (Sub) is in competition with all other axioms and rules (and because subtyping is reflexive it can always be applied, thus even causing endless looping). This problem is easily repaired by integrating subtyping into the axioms and rules directly and in return omitting the explicit (Sub) rule. Then we get a rule system defining a terminating algorithm able to return all types a given term t under a given typing environment Γ is typable to.

5.1 **PolySeq**$^+$ — Removing the (Sub)-Rule

The idea is to use that the subtype relation $\preceq$ is not only reflexive, but also transitive, and to essentially push up the rule (Sub) through all axioms and rules from Figs. 3, 9, and 10 (except (Sub) itself, of course; two adjacent (Sub)-rules simply merge into a single one). For example, in **PolySeq*** we can always replace

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]}\ (\textsc{Cons}) \qquad [\tau] \preceq \tau'}{\Gamma \vdash (t_1 : t_2) :: \tau'}\ (\textsc{Sub})$$

by

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau \qquad \tau \preceq \tau''}{\Gamma \vdash t_1 :: \tau''}\ (\textsc{Sub}) \qquad \dfrac{\Gamma \vdash t_2 :: [\tau] \qquad [\tau] \preceq [\tau'']}{\Gamma \vdash t_2 :: [\tau'']}\ (\textsc{Sub})}{\Gamma \vdash (t_1 : t_2) :: [\tau'']}\ (\textsc{Cons})$$

since $[\tau] \preceq \tau'$ implies $\tau' = [\tau'']$ for some τ'' with $\tau \preceq \tau''$. Hence, the rule (Cons) can be taken over unchanged from **PolySeq*** to our new calculus, which we will call **PolySeq**$^+$.

$$\frac{\tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'}\ (\textsc{Var'}) \qquad \frac{\tau \preceq \tau'}{\Gamma \vdash []_\tau :: [\tau']}\ (\textsc{Nil'})$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]}\ (\textsc{Cons})$$

$$\frac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau_2 \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{case}\ t\ \mathbf{of}\ \{[] \to t_1;\ x_1 : x_2 \to t_2\}) :: \tau_2}\ (\textsc{Case})$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to^\nu \tau_2}\ (\textsc{Abs'}_\nu)_{\nu \in \{\circ, \varepsilon\}}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\ (\textsc{App}_\nu)_{\nu \in \{\circ, \varepsilon\}}$$

$$\frac{\Gamma \vdash t :: \tau \to^\nu \tau \qquad \tau \preceq \tau'}{\Gamma \vdash \mathbf{fix}\ t :: \tau'}\ (\textsc{Fix'}_\nu)_{\nu \in \{\circ, \varepsilon\}}$$

$$\frac{\Gamma \vdash \tau_1 \in \mathsf{Seqable} \qquad \Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2}\ (\textsc{SLet'})$$

Fig. 12 The Typing Rules in **PolySeq**$^+$

For some of the other axioms and rules, changes are necessary. For example, in **PolySeq*** we cannot replace

$$\frac{\dfrac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1 \to \tau_2}\ (\textsc{Abs}) \qquad \tau_1 \to \tau_2 \preceq \tau'}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau'}\ (\textsc{Sub})$$

by a derivation in which (Sub) has moved to the top. So instead we introduce, in **PolySeq**$^+$, new rules

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2' \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to \tau_2'}\ (\textsc{Abs'})$$

and

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2' \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to^\circ \tau_2'}\ (\textsc{Abs'}_\circ)$$

(Note that $\tau_1 \to \tau_2 \preceq \tau'$ means $\tau' = \tau_1' \to^{\nu'} \tau_2'$ for some τ_1', τ_2', ν' with $\tau_1' \preceq \tau_1$, $\tau_2 \preceq \tau_2'$, and $\nu' \in \{\varepsilon, \circ\}$.)

The collection of all typing rules of **PolySeq**$^+$ is given in Fig. 12, where for brevity we again use parameterization of rules. The Seqable and subtyping axiom and rule systems remain unchanged (see Figs. 8 and 11). Typability in **PolySeq*** and in **PolySeq**$^+$ are equivalent, as established next.

Lemma 4 *If $\Gamma \vdash t :: \tau$ holds in **PolySeq***, then $\Gamma \vdash t :: \tau$ holds in **PolySeq**$^+$, and vice versa.*

Proof By induction on typing derivations. □

While **PolySeq**$^+$ does define a terminating algorithm, there is still a high degree of nondeterminism (namely many cases of competition between a pair of rules corresponding to each other, but one introducing ε, the other $\circ$ as annotation), which would typically lead to backtracking in the implementation. Additionally, runs with all possible choices of annotations on the given input Γ and t would be required to gain all suitable refined types. Fortunately, we can do better than that. This is the topic of the next subsection.

5.2 **PolySeq**^{C} — Using Constraints and Conditional Typability

To avoid the production of many trees and several runs with different inputs, we switch to variables as annotations in Γ, t, and τ. This in particular obviates the parameterization of rules as used in **PolySeq**$^+$ to write down a whole rule family as one scheme. Thus, we eliminate any competition between different rules, and allow the interpretation of the resulting rule system as a *deterministic* algorithm.

This solution is realized by the calculus **PolySeq**^{C}, which will be equivalent to **PolySeq**$^+$ (and thus to **PolySeq*** by Lemma 4) in a sense made precise below, but actually states *conditional typability*. We switch to parameterized terms, types, and typing environments — i.e., to terms, types, and typing environments that use variables instead of concrete ε- and $\circ$-annotations. In what follows, parameterized entities are dotted to be distinguishable from concrete ones. Conditional typing judgements are of the form $\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$, where C is a propositional logic formula combining constraints, which are equations and inequations over ε, $\circ$, and annotation variables (typically ν, ν', ...).

For example, the two rules

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to^\circ \tau_2} \ (\textsc{Abs'}_\circ)$$

and

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2 \qquad \tau_1' \preceq \tau_1}{\Gamma \vdash (\lambda x :: \tau_1 . t) :: \tau_1' \to \tau_2} \ (\textsc{Abs'})$$

now become fused into a single one as follows:

$$\frac{\langle \dot{\Gamma}, x :: \dot{\tau_1} \vdash \dot{t} \rangle \Rrightarrow (C_1, \dot{\tau_2}) \qquad \langle \cdot \preceq \dot{\tau_1} \rangle \Rrightarrow (C_2, \dot{\tau_1'})}{\langle \dot{\Gamma} \vdash \lambda x :: \dot{\tau_1} . \dot{t} \rangle \Rrightarrow (C_1 \wedge C_2, \dot{\tau_1'} \to^\nu \dot{\tau_2})} \ (\textsc{Abs}^C)$$

Here, the ν is really an object level variable: it occurs in the "produced" pair $(C_1 \wedge C_2, \dot{\tau_1'} \to^\nu \dot{\tau_2})$ of constraint and type, rather than being a meta level variable used to denote two different rules in a more compact way. From the rule $(\textsc{Abs}^C)$ we also see that not only the actual typing rules from Fig. 12 need to be put into conditional form, but also other axiom and rule systems have to be changed, like the subtyping axiom and rules from Fig. 11. Indeed, since $\dot{\tau_1}$ may contain annotation variables, it is not anymore possible to simply require $\tau_1' \preceq \tau_1$ (or $\tau_1' \preceq \dot{\tau_1}$) as a precondition like in rules $(\textsc{Abs'}_\circ)$ and $(\textsc{Abs'})$ and rely on the axiom and rules from Fig. 11 for checking it. Instead, we cater for the fact that in place of τ_1' we will also want a parameterized entity, along with a constraint (C_2) that appropriately relates the annotation variables in $\dot{\tau_1}$ and $\dot{\tau_1'}$. We do so by providing an auxiliary axiom and rule system later in Fig. 15.

$$\frac{\langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C, \dot\tau')}{\langle \dot\Gamma, x :: \dot\tau \vdash x \rangle \Rrightarrow (C, \dot\tau')}\ (\textsc{Var}^C) \qquad \frac{\langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C, \dot\tau')}{\langle \dot\Gamma \vdash [\,]_{\dot\tau} \rangle \Rrightarrow (C, [\dot\tau'])}\ (\textsc{Nil}^C)$$

$$\frac{\langle \dot\Gamma \vdash \dot t_1 \rangle \Rrightarrow (C_1, \dot\tau) \qquad \langle \dot\Gamma \vdash \dot t_2 \rangle \Rrightarrow (C_2, [\dot\tau']) \qquad \langle \dot\tau = \dot\tau' \rangle \Rrightarrow C_3}{\langle \dot\Gamma \vdash \dot t_1 : \dot t_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, [\dot\tau])}\ (\textsc{Cons}^C)$$

$$\frac{\begin{array}{c}\langle \dot\Gamma \vdash \dot t \rangle \Rrightarrow (C_1, [\dot\tau_1]) \qquad \langle \dot\Gamma \vdash \dot t_1 \rangle \Rrightarrow (C_2, \dot\tau_2) \\ \langle \dot\Gamma, x_1 :: \dot\tau_1, x_2 :: [\dot\tau_1] \vdash \dot t_2 \rangle \Rrightarrow (C_3, \dot\tau_2') \qquad \langle \dot\tau_2 = \dot\tau_2' \rangle \Rrightarrow C_4\end{array}}{\langle \dot\Gamma \vdash \mathbf{case}\ \dot t\ \mathbf{of}\ \{[\,] \to \dot t_1;\ x_1 : x_2 \to \dot t_2\} \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3 \wedge C_4, \dot\tau_2)}\ (\textsc{Case}^C)$$

$$\frac{\langle \dot\Gamma, x :: \dot\tau_1 \vdash \dot t \rangle \Rrightarrow (C_1, \dot\tau_2) \qquad \langle \cdot \preceq \dot\tau_1 \rangle \Rrightarrow (C_2, \dot\tau_1')}{\langle \dot\Gamma \vdash \lambda x :: \dot\tau_1 . \dot t \rangle \Rrightarrow (C_1 \wedge C_2, \dot\tau_1' \to^\nu \dot\tau_2)}\ (\textsc{Abs}^C)$$

$$\frac{\langle \dot\Gamma \vdash \dot t_1 \rangle \Rrightarrow (C_1, \dot\tau_1 \to^\nu \dot\tau_2) \qquad \langle \dot\Gamma \vdash \dot t_2 \rangle \Rrightarrow (C_2, \dot\tau_1') \qquad \langle \dot\tau_1 = \dot\tau_1' \rangle \Rrightarrow C_3}{\langle \dot\Gamma \vdash \dot t_1\ \dot t_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot\tau_2)}\ (\textsc{App}^C)$$

$$\frac{\langle \dot\Gamma \vdash \dot t \rangle \Rrightarrow (C_1, \dot\tau \to^\nu \dot\tau') \qquad \langle \dot\tau = \dot\tau' \rangle \Rrightarrow C_2 \qquad \langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C_3, \dot\tau'')}{\langle \dot\Gamma \vdash \mathbf{fix}\ \dot t \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot\tau'')}\ (\textsc{Fix}^C)$$

$$\frac{\begin{array}{c}\langle \dot\Gamma \vdash \dot t_1 \rangle \Rrightarrow (C_1, \dot\tau_1) \\ \langle \dot\Gamma \vdash \dot\tau_1 \in \mathsf{Seqable} \rangle \Rrightarrow C_2 \qquad \langle \dot\Gamma, x :: \dot\tau_1 \vdash \dot t_2 \rangle \Rrightarrow (C_3, \dot\tau_2)\end{array}}{\langle \dot\Gamma \vdash \mathbf{let!}\ x = \dot t_1\ \mathbf{in}\ \dot t_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot\tau_2)}\ (\textsc{SLet}^C)$$

Fig. 13 The Conditional Typing Rules in $\mathbf{PolySeq}^C$

$$\langle \dot\Gamma \vdash [\dot\tau] \in \mathsf{Seqable} \rangle \Rrightarrow \mathrm{True}\ (\text{C-}\textsc{List}^C)$$

$$\langle \dot\Gamma \vdash (\dot\tau_1 \to^\nu \dot\tau_2) \in \mathsf{Seqable} \rangle \Rrightarrow (\nu = \varepsilon)\ (\text{C-}\textsc{Arrow}^C)$$

$$\langle \alpha^\nu, \dot\Gamma \vdash \alpha \in \mathsf{Seqable} \rangle \Rrightarrow (\nu = \varepsilon)\ (\text{C-}\textsc{Var}^C)$$

Fig. 14 The Conditional Class Membership Axioms for Seqable in $\mathbf{PolySeq}^C$

Similarly, when replacing the two rules ($\textsc{App}_\circ$) and ($\textsc{App}$) from $\mathbf{PolySeq}^+$ by a single one, we need to make use of an auxiliary system stating conditional equality. After all, typing t_1 and t_2 independently could lead to parameterized versions of the types $\tau_1 \to^\nu \tau_2$ and τ_1 in which the "τ_1-parts" differ in the naming of annotation variables. A new auxiliary system then enforces the appropriate equalities via a constraint formula.

Overall, the typing rules for conditional typability are given in Fig. 13, and axioms and rules of auxiliary systems for conditional class membership in Seqable, subtyping, and equality, are shown in Figs. 14–16.

To relate conditional typability to concrete typability on concrete terms, types, and typing environments, we define *annotation substitutions* ϱ that map the parameterized entities $\dot\kappa$ to concrete ones by replacing each annotation variable by one of the concrete annotations ε or $\circ$. We denote the application of an annotation substitution ϱ to $\dot\kappa$ by $\dot\kappa\varrho$. Also, annotation substitutions can be applied to constraints C, denoted by $C\varrho$. If $C\varrho$ is a propositional logic sentence, we write $[\![C\varrho]\!]$ for its value (either True or False). With the help of these tools we define concrete typability in $\mathbf{PolySeq}^C$.

$$\langle \alpha \preceq \cdot \rangle \Rrightarrow (\text{True}, \alpha)\ (\text{S-VAR}_1^C) \qquad \langle \cdot \preceq \alpha \rangle \Rrightarrow (\text{True}, \alpha)\ (\text{S-VAR}_2^C)$$

$$\frac{\langle \cdot \preceq \dot{\sigma_1} \rangle \Rrightarrow (C_1, \dot{\tau_1}) \qquad \langle \dot{\sigma_2} \preceq \cdot \rangle \Rrightarrow (C_2, \dot{\tau_2})}{\langle (\dot{\sigma_1} \to^{\nu} \dot{\sigma_2}) \preceq \cdot \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot{\tau_1} \to^{\nu'} \dot{\tau_2})}\ (\text{S-ARROW}_1^C)$$

$$\frac{\langle \dot{\tau_1} \preceq \cdot \rangle \Rrightarrow (C_1, \dot{\sigma_1}) \qquad \langle \cdot \preceq \dot{\tau_2} \rangle \Rrightarrow (C_2, \dot{\sigma_2})}{\langle \cdot \preceq (\dot{\tau_1} \to^{\nu'} \dot{\tau_2}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge (\nu' \leqslant \nu), \dot{\sigma_1} \to^{\nu} \dot{\sigma_2})}\ (\text{S-ARROW}_2^C)$$

$$\frac{\langle \dot{\tau_1} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau_2})}{\langle [\dot{\tau_1}] \preceq \cdot \rangle \Rrightarrow (C, [\dot{\tau_2}])}\ (\text{S-LIST}_1^C) \qquad \frac{\langle \cdot \preceq \dot{\tau_2} \rangle \Rrightarrow (C, \dot{\tau_1})}{\langle \cdot \preceq [\dot{\tau_2}] \rangle \Rrightarrow (C, [\dot{\tau_1}])}\ (\text{S-LIST}_2^C)$$

Fig. 15 The Conditional Subtyping Axioms and Rules in $\mathbf{PolySeq}^C$

$$\langle \alpha = \alpha \rangle \Rrightarrow \text{True}\ (\text{E-VAR}^C) \qquad \frac{\langle \dot{\tau_1} = \dot{\tau_2} \rangle \Rrightarrow C}{\langle [\dot{\tau_1}] = [\dot{\tau_2}] \rangle \Rrightarrow C}\ (\text{E-LIST}^C)$$

$$\frac{\langle \dot{\sigma_1} = \dot{\tau_1} \rangle \Rrightarrow C_1 \qquad \langle \dot{\sigma_2} = \dot{\tau_2} \rangle \Rrightarrow C_2}{\langle (\dot{\sigma_1} \to^{\nu} \dot{\sigma_2}) = (\dot{\tau_1} \to^{\nu'} \dot{\tau_2}) \rangle \Rrightarrow C_1 \wedge C_2 \wedge (\nu = \nu')}\ (\text{E-ARROW}^C)$$

Fig. 16 The Conditional Equality Axiom and Rules in $\mathbf{PolySeq}^C$

Definition 1 A term t is *(concretely) typable* to τ under Γ in $\mathbf{PolySeq}^C$ if there exist $\dot{\Gamma}$, $\dot{t}$, $\dot{\tau}$, C, and ϱ, such that $\dot{\Gamma}\varrho = \Gamma$, $\dot{t}\varrho = t$, $\dot{\tau}\varrho = \tau$, $[\![C\varrho]\!] = \text{True}$, and $\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$ holds in $\mathbf{PolySeq}^C$.

We can now state equivalence of typability in $\mathbf{PolySeq}^C$and in $\mathbf{PolySeq}^+$.

Theorem 3 *A term t is (concretely) typable to a type τ under a typing environment Γ in $\mathbf{PolySeq}^C$ iff $\Gamma \vdash t :: \tau$ holds in $\mathbf{PolySeq}^+$.*

Proof Via a number of statements about the auxiliary axiom and rule systems from Figs. 14–16, relating them to the ones from Figs. 8 and 11 and to syntactic equality of concrete types. Ultimately, by inductions on typing derivations. Details appear in the technical report [28]. □

6 Putting the Theory to Use

As example of how $\mathbf{PolySeq}^C$ can be used algorithmically for type refinement, we again consider the function *foldl″* from Section 2. The algorithm's input will be the term *foldl″* (in the style of **PolySeq**, in particular with standard type signatures at the binding occurrences of term variables) and the typing environment $\Gamma = \alpha, \beta$. First, we add pairwise distinct variable annotations, $\nu_1, \ldots, \nu_m$, at all type variables in Γ and at all arrows in type signatures in *foldl″*. This manipulation is reflected by putting a dot on top of *foldl″* and of Γ:

$$\begin{aligned} \dot{\Gamma} \quad &= \alpha^{\nu_1}, \beta^{\nu_2} \\ \dot{foldl''} &= \lambda c :: (\alpha \to^{\nu_3} \beta \to^{\nu_4} \alpha).\mathbf{fix}\ (\lambda h :: (\alpha \to^{\nu_5} [\beta] \to^{\nu_6} \alpha).\lambda n :: \alpha.\lambda ys :: [\beta].\cdots) \end{aligned}$$

Then, we use the typing rules of $\mathbf{PolySeq}^C$ from bottom to top to generate a derivation tree for $\dot{foldl''}$ in the typing environment $\dot{\Gamma}$. If there is such a derivation tree (and there *is*, since *foldl″* is typable in the typing environment $\Gamma = \alpha, \beta$ in **PolySeq** and

since we can freely choose fresh annotation variables in different branches of the tree), we can use it to determine C and $\dot{\tau}$ such that $\langle \dot{\Gamma} \vdash \dot{foldl''} \rangle \Rrightarrow (C, \dot{\tau})$ holds in $\mathbf{PolySeq}^C$. The parameterized type $\dot{\tau}$ contains variable annotations $\nu_{m+1}, \ldots, \nu_{m+n}$, and C imposes constraints on $\nu_1, \ldots, \nu_{m+n}$ (and possibly on other annotation variables used only during the typing derivation). Now we determine the annotation substitutions ϱ for which $[\![C\varrho]\!] = \text{True}$ (and which, among others, instantiate all the $\nu_{m+1}, \ldots, \nu_{m+n}$). The applications of these annotation substitutions to $\dot{\tau}$ and to $\dot{\Gamma}$ provide us with all refined types of *foldl''*, along with information about which type variables can be $\circ$-annotated and which cannot. In a last step, we remove types that are not minimal in the obtained set with respect to the subtype relation given by the axiom and rules from Fig. 11, because these types would lead to unnecessary restrictions in the corresponding free theorems. For *foldl''* we end up with the single type $(\alpha \to^\circ \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$ and the information that α can be $\circ$-annotated while β cannot.

The polymorphic calculi considered so far in this paper contain only lists as algebraic data type, but the extension to other polynomial (sum-of-products) types and to base types like Int and Bool is straightforward. $\mathbf{PolySeq}^C$ extended by integers (with addition) and Booleans (with a case-statement and if-then-else), as well as higher-rank polymorphism, has been implemented (source code available at `http://hackage.haskell.org/package/free-theorems-seq-1.0`) and made usable through a web interface (`http://www-ps.iai.uni-bonn.de/cgi-bin/polyseq.cgi`). A screenshot of the output for *foldl''* is shown in Fig. 17.

Let us comment on the respective outputs for all four examples from Section 2, *foldl* and its strictified versions *foldl'*, *foldl''*, and *foldl'''*. The respectively highlighted parts in the free theorems produced indicate that the totality restriction on f remains required for *foldl'*, while the other additional restrictions mentioned in the first paragraph of Section 4 disappear for it. For *foldl''* as input, the totality restriction on f and the restriction that $c = \bot$ iff $c' = \bot$ disappear, but none of the others do, while for *foldl'''* only the restriction that $c = \bot$ iff $c' = \bot$ remains. Regarding *foldl*, all selective-strictness-related restrictions vanish.

For the sake of an example in which there is more than one minimal type, consider the term $t = \lambda x :: ([\alpha] \to \alpha).x$ in typing environment $\Gamma = \alpha$. The minimal refined types turn out to be $([\alpha] \to \alpha) \to ([\alpha] \to \alpha)$ and $([\alpha] \to^\circ \alpha) \to ([\alpha] \to^\circ \alpha)$ (both with $\circ$-annotated α in the typing environment), which are incomparable. This is caused by the contravariance of functions. Consequences of this lack of principal typing (of **PolySeq***), and other issues, are discussed in the next, and final, section.

7 Discussion

We have just seen that **PolySeq*** lacks principal types, even when we are interested only in minimal ones. As a consequence, one may wonder what to do about free theorems then. Typically, the free theorems for all the minimal types will each have their raison d'être. For the example $t = \lambda x :: ([\alpha] \to \alpha).x$ the one minimal type, $([\alpha] \to^\circ \alpha) \to ([\alpha] \to^\circ \alpha)$, leads to the statement that for every strict f and every p and q,

$$f \circ p = q \circ (map\ f) \tag{a}$$

implies

$$f \circ (t\ p) = (t\ q) \circ (map\ f) \tag{b}$$

The term

```
t = (\c::(a -> (b -> a)).
     (fix (\h::(a -> ([b] -> a)).
           (\n::a.
            (\ys::[b].
             (let! z = (c n) in
              (case ys of {[] -> n; x:xs ->
               (let! xs' = xs in (let! x' = x in ((h ((c n) x')) xs')))}))))))))
```

can be typed to the minimal type

```
(forall^o a. (forall b. ((a ->^o (b -> a)) -> (a -> ([b] -> a)))))
```

with the free theorem

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  (forall p :: t1 -> (t3 -> t1).
       forall q :: t2 -> (t4 -> t2).
        (forall x :: t1.
          ((p x = _|_) <=> (q (f x) = _|_))
          && (forall y :: t3. f (p x y) = q (f x) (g y)))
        ==> (((t p = _|_) <=> (t q = _|_))
             && (forall z :: t1.
                  ((t p z = _|_) <=> (t q (f z) = _|_))
                  && (forall v :: [t3]. f (t p z v) = t q (f z) (map g v))))))
```

The normal free theorem for the type without annotations would be:

```
forall t1,t2 in TYPES, f :: t1 -> t2, f strict and total.
 forall t3,t4 in TYPES, g :: t3 -> t4, g strict and total.
  (forall p :: t1 -> t3 -> t1.
       forall q :: t2 -> t4 -> t2.
        ((p = _|_) <=> (q = _|_))
         && (forall x :: t1.
              ((p x = _|_) <=> (q (f x) = _|_))
              && (forall y :: t3. f (p x y) = q (f x) (g y))))
        ==> (((t p = _|_) <=> (t q = _|_))
             && (forall z :: t1.
                  ((t p z = _|_) <=> (t q (f z) = _|_))
                  && (forall v :: [t3]. f (t p z v) = t q (f z) (map g v))))))
```

Fig. 17 Output of the Web Interface for *foldl''* as Input

The other minimal type, $([\alpha] \to \alpha) \to ([\alpha] \to \alpha)$, instead of "(a) implies (b)" leads to the statement "$(p = \bot$ iff $q = \bot) \wedge$ (a) implies $(t\ p = \bot$ iff $t\ q = \bot) \wedge$ (b)". Neither of the two statements is stronger than the other. It is also not obvious how to combine them into a single one. But there might be a way by using that in a certain sense **PolySeq**^{C} (rather than **PolySeq***) *does* have principal minimal types, with symbolic constraints on the annotation variables. For $t = \lambda x :: ([\alpha] \to \alpha).x$ we can actually get $([\alpha] \to^{\nu} \alpha) \to ([\alpha] \to^{\nu'} \alpha)$ with the constraint $\nu' \leqslant \nu$. Instead of instantiating and minimizing this to the two incomparable cases $\nu = \nu' = \circ$ and $\nu = \nu' = \varepsilon$, we could try to derive a more abstract free theorem that encompasses both two concrete statements above. We have not investigated this avenue further, though.

Staying longer on the parameterized level could also help to improve the efficiency of our implementation. Currently, we do not do anything very smart after **PolySeq**^{C} has run: from the variable annotated type and the symbolic constraint we basically generate all instantiations to concrete types that satisfy the constraint, and then find the minimal types in that set of concrete types. Instead, we could try to "symbolically solve" the constraint plus minimization condition. However, the real challenges of scaling our solution up to a practical implementation lie elsewhere. There is a long way to go from the calculus we have studied to a full intermediate language used in a Haskell compiler! We do know how to incorporate term formers for type abstraction and type application in principle (see [28]). One reviewer suggested that it might even become necessary to add facilities for polymorphism over annotation variables inside the term language. Also, we have only considered polynomial data types so far, but not looked at nested and negative types. And certainly, many kinds of engineering problems would arise when trying to adopt our refined type system in a production compiler.

In any case, we see the main benefit of our approach in understanding the sometimes quite subtle potential impact of selective strictness on functions of a given type. It would be useful to see how far our results here can help to "repair" program transformations that are actually implemented in compilers (specifically, the Glasgow Haskell Compiler), but suffer from the presence of selective strictness. In situations like those discussed by Johann and Voigtländer [14], and also Voigtländer [34], how often can refined typing with our system recover a guarantee of semantics preservation? Other future work could be to use **PolySeq*** as a starting point for the automatic generation of counterexamples to free theorems derived without taking selective strictness into account. In [30] we have already done this for general recursion by taking the type system of Launchbury and Paterson [15], discussed briefly towards the end of Section 3 here, as a starting point. In [30] we use the axioms and rules describing the type class Pointed and then purposefully create (sub)terms **fix** $(\lambda x :: \tau.x)$ for τ such that $\Gamma \vdash \tau \in \mathsf{Pointed}$ does *not* hold. Combining this with typed term generation based on intuitionistic proof search [1, 5], we built a generator for counterexamples to free theorems derived without even taking general recursion into account. Since the present paper does for *seq* what Launchbury and Paterson [15] did for *fix*, our results here could provide the base for a transfer of the counterexample generation work to the setting of selective strictness.

Finally, a natural question is whether or not selective strictness should be put under control via the type system in a future version of Haskell (or even removed completely). We have deliberately not taken a stand on this here. What was important to us is that both the costs and benefits of either way should be well understood when making such a decision.

Acknowledgments

Daniel Seidel was supported by the Deutsche Forschungsgemeinschaft under grant VO 1512/1-1. We also thank the reviewers and participants of the workshop where the earlier version [29] was presented, as well as the journal reviewers, for valuable feedback.

References

1. Augustsson, L. (2009). Putting Curry-Howard to work (Invited talk). At *Approaches and Applications of Inductive Programming*.
2. Bernardy, J.-P., Jansson, P., and Claessen, K. (2010). Testing polymorphic properties. In Gordon, A., editor, *European Symposium on Programming, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer-Verlag. doi: 10.1007/978-3-642-11957-6_8.
3. Crary, K. (2005). Logical relations and a case study in equivalence checking. In [22], chapter 6, pages 223–244.
4. Day, N., Launchbury, J., and Lewis, J. (1999). Logical abstractions in Haskell. In Meijer, E., editor, *Haskell Workshop, Proceedings*. Technical Report UU-CS-1999-28, Utrecht University.
5. Dyckhoff, R. (1992). Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807.
6. Fernandes, J., Pardo, A., and Saraiva, J. (2007). A shortcut fusion rule for circular program calculation. In Keller, G., editor, *Haskell Workshop, Proceedings*, pages 95–106. ACM Press. doi: 10.1145/1291201.1291216.
7. Gill, A., Launchbury, J., and Peyton Jones, S. (1993). A short cut to deforestation. In Arvind, editor, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press. doi: 10.1145/165180.165214.
8. Holdermans, S. and Hage, J. (2010). Making "stricterness" more relevant. In Gallagher, J. and Voigtländer, J., editors, *Partial Evaluation and Program Manipulation, Proceedings*, pages 121–130. ACM Press. doi: 10.1145/1706356.1706379.
9. Hudak, P., Hughes, R., Peyton Jones, S., and Wadler, P. (2007). A history of Haskell: Being lazy with class. In Ryder, B. and Hailpern, B., editors, *History of Programming Languages, Proceedings*, pages 12-1–12-55. ACM Press. doi: 10.1145/1238844.1238856.
10. Hughes, R. (1986). Strictness detection in non-flat domains. In Ganzinger, H. and Jones, N., editors, *Programs as Data Objects 1985, Proceedings*, volume 217 of *LNCS*, pages 112–135. Springer-Verlag. doi: 10.1007/3-540-16446-4_7.
11. Johann, P. (2003). Short cut fusion is correct. *J. Funct. Program.*, 13(4):797–814. doi: 10.1017/S0956796802004409.
12. Johann, P. (2005). On proving the correctness of program transformations based on free theorems for higher-order polymorphic calculi. *Math. Struct. Comput. Sci.*, 15(2):201–229. doi: 10.1017/S0960129504004578.
13. Johann, P. and Voigtländer, J. (2004). Free theorems in the presence of seq. In Leroy, X., editor, *Principles of Programming Languages, Proceedings*, volume 39(1) of *ACM SIGPLAN Not.*, pages 99–110. ACM Press. doi: 10.1145/982962.964010.
14. Johann, P. and Voigtländer, J. (2006). The impact of seq on free theorems-based program transformations. *Fundam. Inform.*, 69(1–2):63–102.
15. Launchbury, J. and Paterson, R. (1996). Parametricity and unboxing with unpointed types. In Riis Nielson, H., editor, *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag. doi: 10.1007/3-540-61055-3_38.
16. Mycroft, A. (1980). The theory and practice of transforming call-by-need into call-by-value. In Robinet, B., editor, *Colloque International sur la Programmation, Proceedings*, volume 83 of *LNCS*, pages 269–281. Springer-Verlag. doi: 10.1007/3-540-09981-6_19.

17. Nielson, F. and Riis Nielson, H. (1999). Type and effect systems. In Olderog, E.-R. and Steffen, B., editors, *Festschrift to Hans Langmaack: Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*, pages 114–136. Springer-Verlag. doi: 10.1007/3-540-48092-7_6.
18. Oliveira, B., Schrijvers, T., and Cook, W. (2010). EffectiveAdvice: Disciplined advice with explicit effects. In Jézéquel, J.-M. and Südholt, M., editors, *Aspect-Oriented Software Development, Proceedings*, pages 109–120. ACM Press. doi: 10.1145/1739230.1739244.
19. Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.
20. Peyton Jones, S., editor (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.
21. Pierce, B. (2002). *Types and Programming Languages*. MIT Press.
22. Pierce, B., editor (2005). *Advanced Topics in Types and Programming Languages*. MIT Press.
23. Pitts, A. (2005). Typed operational reasoning. In [22], chapter 7, pages 245–289.
24. Plasmeijer, R. and van Eekelen, M. (2002). Clean version 2.1 language report. `http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf`.
25. Reynolds, J. (1974). Towards a theory of type structure. In Robinet, B., editor, *Colloque sur la Programmation, Proceedings*, volume 19 of *LNCS*, pages 408–423. Springer-Verlag.
26. Reynolds, J. (1983). Types, abstraction and parametric polymorphism. In Mason, R., editor, *Information Processing, Proceedings*, pages 513–523. Elsevier.
27. Schmidt, D. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon.
28. Seidel, D. and Voigtländer, J. (2009a). Taming selective strictness. Technical Report TUD-FI09-06, Technische Universität Dresden. `http://www.iai.uni-bonn.de/~jv/TUD-FI09-06.pdf`.
29. Seidel, D. and Voigtländer, J. (2009b). Taming selective strictness. In Dosch, W. and Hanus, M., editors, *Arbeitstagung Programmiersprachen, Proceedings*, volume 154 of *LNI*, pages 2916–2930. Gesellschaft für Informatik.
30. Seidel, D. and Voigtländer, J. (2010). Automatically generating counterexamples to naive free theorems. In Blume, M. and Vidal, G., editors, *Functional and Logic Programming, Proceedings*, volume 6009 of *LNCS*, pages 175–190. Springer-Verlag.
31. Svenningsson, J. (2002). Shortcut fusion for accumulating parameters & zip-like functions. In Peyton Jones, S., editor, *International Conference on Functional Programming, Proceedings*, volume 37(9) of *ACM SIGPLAN Not.*, pages 124–132. ACM Press. doi: 10.1145/583852.581491.
32. Voigtländer, J. (2008a). Much ado about two: A pearl on parallel prefix computation. In Wadler, P., editor, *Principles of Programming Languages, Proceedings*, volume 43(1) of *ACM SIGPLAN Not.*, pages 29–35. ACM Press. doi: 10.1145/1328897.1328445.
33. Voigtländer, J. (2008b). Proving correctness via free theorems: The case of the destroy/build-rule. In Glück, R. and de Moor, O., editors, *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 13–20. ACM Press. doi: 10.1145/1328408.1328412.
34. Voigtländer, J. (2008c). Semantics and pragmatics of new shortcut fusion rules. In Garrigue, J. and Hermenegildo, M., editors, *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag. doi: 10.1007/978-

3-540-78969-7_13.

35. Voigtländer, J. (2009a). Bidirectionalization for free! In Pierce, B., editor, *Principles of Programming Languages, Proceedings*, volume 44(1) of *ACM SIGPLAN Not.*, pages 165–176. ACM Press. doi: 10.1145/1594834.1480904.
36. Voigtländer, J. (2009b). Free theorems involving type constructor classes. In Tolmach, A., editor, *International Conference on Functional Programming, Proceedings*, volume 44(9) of *ACM SIGPLAN Not.*, pages 173–184. ACM Press. doi: 10.1145/1631687.1596577.
37. Voigtländer, J., Hu, Z., Matsuda, K., and Wang, M. (2010). Combining syntactic and semantic bidirectionalization. In Weirich, S., editor, *International Conference on Functional Programming, Proceedings*, volume 45(9) of *ACM SIGPLAN Not.*, pages 181–192. ACM Press. doi: 10.1145/1932681.1863571.
38. Wadler, P. (1989). Theorems for free! In MacQueen, D., editor, *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press. doi: 10.1145/99370.99404.
39. Wadler, P. and Blott, S. (1989). How to make *ad-hoc* polymorphism less *ad hoc*. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press. doi: 10.1145/75277.75283.