Taming Selective Strictness

Daniel Seidel* and Janis Voigtländer
Technische Universität Dresden, 01062 Dresden, Germany
{seideld,voigt}@tcs.inf.tu-dresden.de

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 restrengthening them. By a refined type system which tracks the use of strict evaluation, we rule out unnecessary restrictions that otherwise emerge from the general suspicion that strict evaluation may be used at any point. Additionally, we provide an implemented algorithm determining all refined types for a given term.

1 Introduction

Free theorems [Wad89] are a useful proof tool in lazy functional languages like Haskell, in particular for verifying program transformations [GLP93, Joh03, Voi08b], but also for other interesting results [Voi08a, Voi09a, Voi09b]. Initially, free theorems have been investigated in the pure polymorphic lambda calculus, additionally taking the influence of general recursion into account. But modern languages like Haskell and Clean extend the pure polymorphic lambda calculus not only by a fixpoint combinator; they additionally allow selective strictness. Selective strict evaluation is in particular desirable to avoid space leaks that are otherwise likely to arise in lazy languages. A disadvantage is the weakening of relational parametricity, and hence the free theorems based on it.

Consider the well-known Haskell Prelude function foldl, its strict variant foldl' (in the Haskell standard library `Data.List`), and functions foldl'' and foldl''' which force strict evaluation at rather arbitrary points, with implementations as shown in Figure 1. Strict evaluation is expressed via seq, which evaluates its first argument, returns the second argument if that evaluation is successful, and otherwise fails. The fixpoint combinator $\mathit{fix} :: \forall\alpha.(\alpha \to \alpha) \to \alpha$ expresses general recursion.

All four functions are of type $\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$, and the corresponding free theorem, ignoring potential strict evaluation, states that

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

for appropriately typed c, c', n, xs and strict f, g such that $\forall x, y.\ f\ (c\ x\ y) = c'\ (f\ x)\ (g\ y)$.

*This author was supported by the DFG under grant VO 1512/1-1.

```
$foldl\ c = fix$
  $(\lambda h\ n\ ys \rightarrow$
      **case** $ys$ **of**
        $[\,]\quad \rightarrow n$
        $x : xs \rightarrow$
          **let** $n' = c\ n\ x$ **in** $h\ n'\ xs)$

$foldl'\ c = fix$
  $(\lambda h\ n\ ys \rightarrow$
      **case** $ys$ **of**
        $[\,]\quad \rightarrow n$
        $x : xs \rightarrow$
          **let** $n' = c\ n\ x$ **in** $seq\ n'\ (h\ n'\ xs))$

$foldl''\ c = fix$
  $(\lambda h\ n\ ys \rightarrow$
      $seq\ (c\ n)$
        $($**case** $ys$ **of**
          $[\,]\quad \rightarrow n$
          $x : xs \rightarrow seq\ xs$
            $(seq\ x$
              $($**let** $n' = c\ n\ x$ **in** $h\ n'\ xs))))$

$foldl'''\ c = seq\ c\ (fix$
  $(\lambda h\ n\ ys \rightarrow$
      **case** $ys$ **of**
        $[\,]\quad \rightarrow n$
        $x : xs \rightarrow$
          **let** $n' = c\ n\ x$ **in** $h\ n'\ xs))$
```

Figure 1: Variants of *foldl* with Different Uses of *seq*

Taking strict evaluation into account, the situation changes and additional preconditions become necessary. For example, for the Haskell function $foldl'$ the free theorem as stated above does not hold.

Consider equation (1) with the instantiations

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

Regarding *foldl* everything is fine, but for the strict $foldl'$ we get $True = \bot$. In that case, to require equivalence it suffices to restrict f to be total ($f\ x \neq \bot$ for every $x \neq \bot$), but if we regard the functions $foldl''$ and $foldl'''$, for which the just given instantiation does not break the free theorem, we will encounter the necessity of further restrictions. Consider each of the following instantiations:

$f = id$	$g = t_1$	$c = t_2$	$c' = t_2$	$n = True$	$xs = [False]$
$f = id$	$g = id$	$c = t_3$	$c' = t_4$	$n = False$	$xs = [\,]$
$f = id$	$g = id$	$c = \bot$	$c' = \lambda x \rightarrow \bot$	$n = False$	$xs = [\,]$

where $t_1 = \lambda x \rightarrow \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$, $t_2 = \lambda x\ y \rightarrow \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } y$, $t_3 = \lambda x\ y \rightarrow \textbf{if } x \textbf{ then } \mathit{True} \textbf{ else } \bot$ and $t_4 = \lambda x \rightarrow \textbf{if } x \textbf{ then } \lambda y \rightarrow \mathit{True} \textbf{ else } \bot$.

For each of these instantiations equation (1) holds for *foldl* and $foldl'$, but the first and the second instantiation break the equation for $foldl''$, while the last instantiation breaks the equation for $foldl'''$. All three failures are caused by different uses of *seq*, which force different restrictions. Only the strict evaluation of the list xs causes no additional restriction.

Hence, we see that not *whether* strict evaluation is used somewhere, far more *where* it is used determines the necessity and the quality of restrictions. So a natural question is as follows: How can we express detailed information about the use of strict evaluation such that we can reduce the restrictions on free theorems?

Since free theorems depend only on the *type* of a term, the information has to be part of the type signature. Hence, we track strict evaluation in the type of a term and thus will be able to determine based on the type whether strictness-caused semantic changes, and hence weakening of parametricity, may arise. It was already attempted to do so when seq was first introduced into Haskell (version 1.3). The type class `Eval` was introduced to make strict evaluation and the resulting limitations with respect to parametricity explicit from the type. But the type class approach presumes that all necessary restrictions can be read off from constraints on type variables. And this is not actually the case for all restrictions arising from selective strict evaluation. For example, $foldl'''$ from Figure 1 would incur no `Eval`-constraint at all, but as seen above, the use of seq on c *does* cause problems. The (so far unrecognized) 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 in [JV04] regarding the special care that is required precisely for the interaction between selective strictness, parametricity, and function types. Even if we consider function types to not in general be in `Eval`, and instead constrain their membership more specifically by allowing type class restrictions on compound types[1], the problems of the type class approach remain. Consider a function f :: `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.

To avoid these problems, we make a different choice for tracking selective strictness. Namely, we provide special annotations at quantification over type variables but also at function types. This leads to a clear correspondence to the impact of strict evaluation on free theorems. Combining the insights of [JV04] with ideas of [LP96] regarding taming general recursion, we present a calculus that allows for 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://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/polyseq.cgi`.

2 Standard Parametricity

We start from a standard denotational semantics for a polymorphic lambda calculus that corresponds to Haskell. Towards the end of this section we describe the notion of relational parametricity for that calculus. We are then on a par with the relevant results of [JV04].

The syntax of types and terms is given by

$$\begin{aligned} \tau &::= \alpha \mid [\tau] \mid \tau \to \tau \mid \forall\alpha.\tau \\ t &::= x \mid [\,]_\tau \mid t : t \mid \mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t;\ x : x \to t\} \mid \\ &\qquad \lambda x :: \tau.t \mid t\,t \mid \Lambda\alpha.t \mid t_\tau \mid \mathbf{fix}\ t \mid \mathbf{let!}\ x = t\ \mathbf{in}\ t \end{aligned}$$

[1]This is not allowed in Haskell 98, but as an extension in GHC, enabled by `-XFlexibleContexts`.

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

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

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

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: (\tau_1 \to \tau_2)} \ (\textsc{Abs}) \qquad \frac{\alpha, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda\alpha.t) :: (\forall\alpha.\tau)} \ (\textsc{TAbs})$$

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

Figure 2: Typing Rules in **PolySeq** (and later **PolySeq***), Part 1

$$\frac{\Gamma \vdash t :: (\forall\alpha.\tau_1)}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]} \ (\textsc{TApp}) \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash t_2 :: \tau_2}{\Gamma \vdash (\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2) :: \tau_2} \ (\textsc{SLet})$$

Figure 3: Typing Rules in **PolySeq**, Part 2

where α ranges over type variables, and x over term variables. We include lists as representative for algebraic data types. Note that the calculus is explicitly typed and that type abstraction and application are explicit in the syntax as well. 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.

Figures 2 and 3 give the typing rules for the calculus. Standard conventions apply here. In particular, typing environments Γ take the form $\alpha_1, \ldots, \alpha_k, x_1 :: \tau_1, \ldots, x_l :: \tau_l$ with distinct α_i and x_j, where all free variables occurring in a τ_j have to be among the listed type variables.

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

$$\begin{aligned}&\mathbf{fix}\ (\lambda m :: \tau.\Lambda\alpha.\Lambda\beta.\lambda h :: \alpha \to \beta.\lambda l :: [\alpha].\\ &\qquad \mathbf{case}\ l\ \mathbf{of}\ \{[\,] \to [\,]_\beta\,;\ x : y \to (h\ x) : ((m_\alpha)_\beta\ h\ y)\})\,.\end{aligned}$$

The denotational semantics interprets types as *pointed complete partial orders* (for short, *pcpos*; least element always denoted $\bot$). The definition in Figure 4, assuming θ to be a mapping from type variables to pcpos, is entirely standard. The operation $\mathit{lift}_\bot$ 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. To avoid confusion, the original elements are tagged, i.e., $\mathit{lift}_\bot\, S = \{\bot\} \cup \{\lfloor s \rfloor \mid s \in S\}$. 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 to provide for infinite lists. The function space lifted in the definition of $[\![\tau_1 \to \tau_2]\!]_\theta$ is the one of monotonic and

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

Figure 4: Semantics of Types

$$
\begin{aligned}
[\![x]\!]_{\theta,\sigma} &= \sigma(x)\\
[\![[\,]_\tau]\!]_{\theta,\sigma} &= \lfloor[\,]\rfloor\\
[\![t_1 : t_2]\!]_{\theta,\sigma} &= \lfloor[\![t_1]\!]_{\theta,\sigma} : [\![t_2]\!]_{\theta,\sigma}\rfloor\\
[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\ ;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma} &= \\
&\begin{cases} [\![t_1]\!]_{\theta,\sigma} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor[\,]\rfloor\\ [\![t_2]\!]_{\theta,\sigma[x_1\mapsto a,\, x_2\mapsto b]} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor a : b\rfloor\\ \bot & \text{if } [\![t]\!]_{\theta,\sigma} = \bot \end{cases}\\
[\![\lambda x :: \tau.t]\!]_{\theta,\sigma} &= \lfloor\lambda a.[\![t]\!]_{\theta,\sigma[x\mapsto a]}\rfloor
\end{aligned}
$$

$$
\begin{aligned}
[\![t_1\ t_2]\!]_{\theta,\sigma} &= [\![t_1]\!]_{\theta,\sigma}\ \$\ [\![t_2]\!]_{\theta,\sigma}\\
[\![\Lambda\alpha.t]\!]_{\theta,\sigma} &= \lambda D.[\![t]\!]_{\theta[\alpha\mapsto D],\sigma}\\
[\![t_\tau]\!]_{\theta,\sigma} &= [\![t]\!]_{\theta,\sigma}\ [\![\tau]\!]_\theta\\
[\![\mathbf{fix}\ t]\!]_{\theta,\sigma} &= \textstyle\bigsqcup_{n\geq 0}\ ([\![t]\!]_{\theta,\sigma}\ \$)^n\ \bot\\
[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]_{\theta,\sigma} &= \\
&\begin{cases} [\![t_2]\!]_{\theta,\sigma[x\mapsto a]} & \text{if } [\![t_1]\!]_{\theta,\sigma} = a \neq \bot\\ \bot & \text{if } [\![t_1]\!]_{\theta,\sigma} = \bot \end{cases}
\end{aligned}
$$

Figure 5: Semantics of Terms

continuous maps between $[\![\tau_1]\!]_\theta$ and $[\![\tau_2]\!]_\theta$, ordered point-wise. Finally, polymorphic types are interpreted as sets of functions from pcpos to values restricted as in the last line of Figure 4, and again ordered point-wise (i.e., $g_1 \sqsubseteq g_2$ iff for every pcpo D, $g_1\ D \sqsubseteq g_2\ D$).

The semantics of terms in Figure 5 is also standard. It uses λ for denoting anonymous functions, and the following operator:

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

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

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 Figure 6, 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

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

where again the greatest fixpoint is taken. For two pcpos D_1 and D_2, $\mathit{Rel}(D_1, D_2)$ collects 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 $(\bot,\bot)$ and closure under suprema. A 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 $\Delta_{\tau_1\to\tau_2,\rho}$ serves the purpose of ensuring that bottom-reflection is preserved throughout the logical relation.

$$\begin{array}{ll}
\Delta_{\alpha,\rho} & = \rho(\alpha) \\
\Delta_{[\tau],\rho} & = \mathit{list}\, \Delta_{\tau,\rho} \\
\Delta_{\tau_1 \to \tau_2,\rho} & = \{(f,g) \mid f = \bot \text{ iff } g = \bot,\ \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f \ \$ \ a, g \ \$ \ b) \in \Delta_{\tau_2,\rho}\} \\
\Delta_{\forall \alpha.\tau,\rho} & = \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}(D_1,D_2).(u\ D_1, v\ D_2) \in \Delta_{\tau,\rho[\alpha \mapsto \mathcal{R}]}\}
\end{array}$$

Figure 6: Standard 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 2.1 *If ρ maps only to relations in Rel, then $\Delta_{\tau,\rho} \in \mathit{Rel}$.*

The lemma is crucial for then proving the following theorem [SV09, Appendix A].

Theorem 2.2 (Parametricity) *If $\Gamma \vdash t :: \tau$, then for every θ_1, θ_2, ρ, σ_1, and σ_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, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$,*

we have $([\![t]\!]_{\theta_1,\sigma_1}, [\![t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}$.

3 Refining the Calculus

If we recall the fold functions from the introduction and the "seq-ignoring" version of the corresponding free theorem, stated in equation (1), we can compare that version with the "seq-safe" version arising from Theorem 2.2. 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.

As already mentioned in the introduction and also apparent from the proof of Theorem 2.2, these additional restrictions arise from different potential uses of strict evaluation and are each respectively only necessary if strict evaluation is used at a special place. Hence, it is reasonable to make strict evaluation (and the place of its use) visible from the type of a term. In particular, the use of strict evaluation on elements of a type should be visible for type variables and function types. Strict 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 whose elements are strictly evaluated from those whose elements are not, or more precisely whose elements are (are not) allowed to be strictly evaluated. Therefore we introduce marks ε and $\circ$ at occurrences of the type constructor $\to$ as well as at type variables in the typing environment. A mark ε signifies that strict evaluation is allowed on the entity in question, whereas a mark $\circ$ prevents the use of strict evaluation at a certain place. We also need to keep track of the distinction for type variables when they get quantified. This is achieved by introducing two different quantifiers, $\forall^{\varepsilon}$ and

$\forall^\circ$. Recalling the example *foldl''* from the introduction, one of its refined types would be $\forall^\circ\alpha.\forall^\varepsilon\beta.(\alpha \to^\circ \beta \to^\varepsilon \alpha) \to^\varepsilon \alpha \to^\varepsilon [\beta] \to^\varepsilon \alpha$.

Using the rule system in Figure 7 we define exactly the types whose elements we allow to be strictly evaluated, by collecting them in the class Seqable. Note that $(\text{C-TABS}_\nu)_{\nu\in\{\circ,\varepsilon\}}$ represents two rules. The maybe surprising α^ε in the premise of $(\text{C-TABS}_\circ)$ lets trivial cases like $\forall^\circ\alpha.\alpha$ be in Seqable. Since that type is inhabited only by $\bot$, a term using strict evaluation on an element of it can always be replaced by $\mathbf{fix}\ id$ itself. Hence, we do not care to prevent, or keep track of, that particular use of strict evaluation.

Having an explicit way to describe which types support selective strict evaluation, we can restrict the typing rules (SLET) and (TAPP) to these types. For convenience, in the remainder of the paper we take ε to be the invisible mark and drop it. The new rules are as follows:

$$\frac{\Gamma \vdash \tau_2 \in \mathsf{Seqable} \qquad \Gamma \vdash t :: (\forall\alpha.\tau_1)}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\ (\text{TAPP'})$$

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

The other typing rules of **PolySeq**, shown in Figure 2, remain unchanged, but we add $\circ$-marked versions $(\text{ABS}_\circ)$, $(\text{APP}_\circ)$, $(\text{TABS}_\circ)$, $(\text{FIX}_\circ)$, and $(\text{TAPP}_\circ)$, with all explicit occurrences of type variables in the typing environment, as well as all explicit occurrences of $\to$ and $\forall$, marked by $\circ$. This extension is straightforward and we show only an example:

$$\frac{\Gamma \vdash t :: (\forall^\circ\alpha.\tau_1)}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\ (\text{TAPP}_\circ)$$

The last rule required for the extended calculus is

$$\frac{\Gamma \vdash t :: \tau_1 \qquad \tau_1 \preceq \tau_2}{\Gamma \vdash t :: \tau_2}\ (\text{SUB})$$

and it needs some explanation. The motivation for it is a subtype relation. 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 will be typable to the second one as well. But, for example, the **PolySeq**-term $\lambda f :: \tau_1 \to \tau_2.\mathbf{let!}\ x = f\ \mathbf{in}\ [\,]$ is only typable to the second one. So whether or not strict evaluation is allowed at arguments of a function determines the direction of the subtype relationship. The subtyping system presented in Figure 8 goes a bit further and restricts the relation thus that a Seqable supertype has only Seqable subtypes. We can think of this as follows: the set of functions on which we do allow strict evaluation is a subtype of the set of functions on which we do not. The rules are written as parameterized rule families, where $\{\circ, \varepsilon\}$ is the ordered set of marks with $\circ < \varepsilon$.

The rule systems just described set up a new calculus **PolySeq***. The definition of a *mark eraser* $|\cdot|$, removing all $\circ$-marks when applied to a term, type, or typing environment, enables us to take over the term and type semantics from **PolySeq** by using $[\![|\cdot|]\!]$, and also allows us to prove the set of typable terms in **PolySeq*** and **PolySeq** to be equivalent in the sense of the following lemma.

$$\Gamma \vdash [\tau] \in \mathsf{Seqable}\ (\text{C-List}) \qquad \Gamma \vdash (\tau_1 \to^{\varepsilon} \tau_2) \in \mathsf{Seqable}\ (\text{C-Arrow})$$

$$\frac{\alpha^{\varepsilon} \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Seqable}}\ (\text{C-Var}) \qquad \frac{\alpha^{\varepsilon}, \Gamma \vdash \tau \in \mathsf{Seqable}}{\Gamma \vdash (\forall \alpha^{\nu}.\tau) \in \mathsf{Seqable}}\ (\text{C-TAbs}_{\nu})_{\nu \in \{\circ,\varepsilon\}}$$

Figure 7: Class Membership Rules for Seqable in **PolySeq***

$$\alpha \preceq \alpha\ (\text{S-Var})$$

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

$$\frac{\tau_1 \preceq \tau_2}{(\forall^{\nu}\alpha.\tau_1) \preceq (\forall^{\nu'}\alpha.\tau_2)}\ (\text{S-All}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu \leqslant \nu'} \qquad \frac{\tau \preceq \tau'}{[\tau] \preceq [\tau']}\ (\text{S-List})$$

Figure 8: Subtyping Rules in **PolySeq***

Lemma 3.1 *If Γ, t, and τ are such that $\Gamma \vdash t :: \tau$ in **PolySeq**, then $\Gamma \vdash t :: \tau$ in **PolySeq***. Conversely, if Γ, t, τ are such that $\Gamma \vdash t :: \tau$ in **PolySeq***, then $|\Gamma| \vdash |t| :: |\tau|$ in **PolySeq**.*

The main reason for restricting strict evaluation 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 the resulting restrictions on free theorems. Hence, we allow the relational actions for $\to^{\circ}$ and $\forall^{\circ}$ to forget about bottom-reflection, and define them as

$$\begin{aligned}
\Delta_{\tau_1 \to^{\circ} \tau_2,\rho} &= \{(f,g) \mid \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau_2,\rho}\}\,, \\
\Delta_{\forall^{\circ}\alpha.\tau,\rho} &= \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^{\circ}(D_1,D_2).(u\ D_1, v\ D_2) \in \Delta_{\tau,\rho[\alpha \mapsto \mathcal{R}]}\}\,,
\end{aligned}$$

where $\mathit{Rel}^{\circ}(D_1,D_2)$ is the set of all strict and continuous (*not necessarily bottom-reflecting*) relations between the pcpos D_1 and D_2. The other relational actions remain as in **PolySeq** (cf. Figure 6).

The resulting logical relation 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-marked in the typing environment. This enables us to state a refined parametricity theorem for **PolySeq*** that allows for stronger free theorems if we localize the use of strict evaluation.

Theorem 3.2 (Parametricity) *If $\Gamma \vdash t :: \tau$ in **PolySeq***, then for every θ_1, θ_2, ρ, σ_1, and σ_2 such that*

- *for every α° occurring in Γ, $\rho(\alpha) \in \mathit{Rel}^{\circ}(\theta_1(\alpha), \theta_2(\alpha))$,*
- *for every α^{ε} occurring in Γ, $\rho(\alpha) \in \mathit{Rel}(\theta_1(\alpha), \theta_2(\alpha))$, and*
- *for every $x :: \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$,*

we have $([\![|t|]\!]_{\theta_1,\sigma_1}, [\![|t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}$.

The proof is very similar to the standard proof of Theorem 2.2, and is given in [SV09, Theorem 3.8]. Where needed, membership in the Seqable-class assures bottom-reflection. Since the subtype relation of two types guarantees also that the logical relations of these types are subsets of each other, i.e. $\tau_1 \preceq \tau_2 \Rightarrow \Delta_{\tau_1,\rho} \subseteq \Delta_{\tau_2,\rho}$ for each appropriate ρ, the inductive case for the (SUB) rule is easily proved.

If we regard foldl'' from the introduction with the refined type $\forall^{\circ}\alpha.\forall\beta.(\alpha \rightarrow^{\circ} \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$ once again, Theorem 3.2 tells us 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 free theorem for the completely unmarked type as stated at the beginning of the current section.

4 PolySeq^C — Obtaining all Permissible Types

The calculus **PolySeq*** allows refined typing for all terms typable in **PolySeq**. The final aim is to provide automatic type refinement for given terms with standard (**PolySeq**) typings. Hence, the intended use of **PolySeq*** will be to input a **PolySeq** term t, in particular without (ε- or) $\circ$-marks, and to find either all, or better all minimal (in the sense of strongest free theorems, which corresponds to the minimal logical relations) permissible types that t, for some concrete setting of marks in its syntactic type components (e.g., at occurrences of the empty list or in λ-abstractions), is typable to in **PolySeq***. Or, more generally, the same setting with given t closed under a (fixed up to ε- vs. $\circ$-marks) given typing environment Γ has to be handled.

Unfortunately, **PolySeq*** is not suitable for an algorithmic use in its current form. The rule (SUB) is in competition with all other rules, and because subtyping is reflexive it can always be applied and thus cause endless looping. This problem is easily repaired by integrating subtyping into the rules directly and in return omitting the explicit (SUB) rule. Then we have 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. But there will still be ambiguity between typing derivations, since sometimes we have the choice between two rules, one introducing ε, the other $\circ$ as mark, which makes backtracking necessary. Additionally, runs with all possible choices of marks on the given input Γ and t would be required to gain all suitable refined types.

Alternatively, avoiding the production of many trees and several runs with different inputs, we can switch to variables as marks in Γ and t. This will in particular obviate the parameterization of rules as used in **PolySeq*** to write down a whole rule family as one scheme. Thereby, we eliminate any competition between different rules, allowing the interpretation of the resulting rule system as a *deterministic* algorithm.

This solution is realized by the calculus **PolySeq**^{C}, which is equivalent to **PolySeq*** in a sense made precise below, but actually states *conditional typability*. We switch to parameterized terms, types, and typing environments that use variables instead of concrete ε- and $\circ$-marks. In what follows, parameterized entities are dotted to be distinguishable from concrete ones. The conditional typing rules are of the form $\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$, where

C is a propositional logic formula combining constraints on mark variables ν. The typing rules for conditional typability are given in Figure 9, and rules of auxiliary systems stating conditional class membership in Seqable, subtyping, and equality, are shown in Figures 10–12. To relate conditional typability to concrete typability on concrete terms, types, and typing environments, we define *mark replacements* ϱ that map the parameterized entities $\dot{\kappa}$ to concrete ones by replacing each mark variable by one (and for different occurrences of the same variable, the same) of the concrete marks ε or $\circ$. We denote the application of a mark replacement ϱ to $\dot{\kappa}$ by $\dot{\kappa}_\varrho$. Furthermore, mark replacements can be applied to constraints C, denoted by C_ϱ. By convention, if C_ϱ is a propositional logic sentence, it is identified with its value, i.e. either True or False.

With the help of these tools we define concrete typability in **PolySeq**^{C}.

Definition 4.1 *A term t is* (concretely) typable *to τ under Γ in **PolySeq**^{C} if there exist $\dot{\Gamma}$, $\dot{t}$, $\dot{\tau}$, C, and ϱ, such that $\dot{\Gamma}_\varrho = \Gamma$, $\dot{t}_\varrho = t$, $\dot{\tau}_\varrho = \tau$, $C_\varrho =$ True, and $\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$ in **PolySeq**^{C}.*

We can now state equivalence of typability in **PolySeq*** and **PolySeq**^{C}. The proof can be found in [SV09, Lemma 4.1 and Theorem 5.3].

Theorem 4.2 *A term t is (concretely) typable to a type τ under a typing environment Γ in **PolySeq**^{C} iff it is typable to τ under Γ in **PolySeq***.*

As example of how **PolySeq**^{C} can be used algorithmically for type refinement, we again consider the function *foldl*$''$ from the introduction. The algorithm's input will be the term *foldl*$''$ (in the style of **PolySeq**, in particular with standard type annotations at the binding occurrences of term variables). Since *foldl*$''$ is closed, the initial typing environment is empty. First, we add pairwise distinct variable marks, $\nu_1, \ldots, \nu_m$, at all $\forall$-quantifiers and arrows in type annotations in *foldl*$''$. This manipulation is reflected by putting a dot on top of *foldl*$''$. Then, we use the typing rules of **PolySeq**^{C} backwards to generate a derivation tree for $\dot{\textit{foldl}}''$ in the empty typing environment. If there is such a derivation tree (and since *foldl*$''$ is typable in **PolySeq**, there is), we can use it to determine C and $\dot{\tau}$ such that $\langle \vdash \dot{\textit{foldl}}'' \rangle \Rrightarrow (C, \dot{\tau})$ in **PolySeq**^{C}. The parameterized type $\dot{\tau}$ contains variable marks $\nu_{m+1}, \ldots, \nu_{m+n}$, and C imposes constraints on $\nu_1, \ldots, \nu_{m+n}$ (and possibly on other mark variables used only during the typing derivation). Now we determine the mark replacements ϱ for which C_ϱ is True (and which, among others, instantiate all the $\nu_{m+1}, \ldots, \nu_{m+n}$). The applications of these mark replacements to $\dot{\tau}$ provide us all refined types of *foldl*$''$. In a last step, we remove types that are not minimal in this set with respect to the subtype relation given by the rules from Figure 8, because these types would lead to unnecessary restrictions in the corresponding free theorems. For *foldl*$''$ we end up with the single, already known type $\forall^\circ\alpha.\forall\beta.(\alpha \to^\circ \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$, but now it has been generated automatically.

For the sake of an example in which there is more than one minimal type, consider the term $t = \Lambda\alpha.\lambda x :: ([\alpha] \to \alpha).x$. Its minimal refined types are $\forall^\circ\alpha.(([\alpha] \to \alpha) \to ([\alpha] \to \alpha))$ and $\forall^\circ\alpha.(([\alpha] \to^\circ \alpha) \to ([\alpha] \to^\circ \alpha))$, which are incomparable.

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

$$\frac{\langle \dot{\Gamma} \vdash \dot{t}_1 \rangle \Rrightarrow (C_1, \dot{\tau}) \qquad \langle \dot{\Gamma} \vdash \dot{t}_2 \rangle \Rrightarrow (C_2, [\dot{\tau}']) \qquad \langle \dot{\tau} = \dot{\tau}' \rangle \Rrightarrow C_3}{\langle \dot{\Gamma} \vdash \dot{t}_1 : \dot{t}_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, [\dot{\tau}])}\ (\textsc{Cons}^C)$$

$$\frac{\begin{array}{c}\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C_1, [\dot{\tau}_1]) \qquad \langle \dot{\Gamma} \vdash \dot{t}_1 \rangle \Rrightarrow (C_2, \dot{\tau}_2) \\ \langle \dot{\Gamma}, x_1 :: \dot{\tau}_1, x_2 :: [\dot{\tau}_1] \vdash \dot{t}_2 \rangle \Rrightarrow (C_3, \dot{\tau}_2') \qquad \langle \dot{\tau}_2 = \dot{\tau}_2' \rangle \Rrightarrow C_4\end{array}}{\langle \dot{\Gamma} \vdash \textbf{case}\ \dot{t}\ \textbf{of}\ \{[\,] \to \dot{t}_1\ ;\ x_1 : x_2 \to \dot{t}_2\} \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3 \wedge C_4, \dot{\tau}_2)}\ (\textsc{LCase}^C)$$

$$\frac{\langle \dot{\Gamma}, x :: \dot{\tau}_1 \vdash \dot{t} \rangle \Rrightarrow (C_1, \dot{\tau}_2) \qquad \langle \cdot \preceq \dot{\tau}_1 \rangle \Rrightarrow (C_2, \dot{\tau}_1')}{\langle \dot{\Gamma} \vdash \lambda x :: \dot{\tau}_1 . \dot{t} \rangle \Rrightarrow (C_1 \wedge C_2, \dot{\tau}_1' \to^{\nu} \dot{\tau}_2)}\ (\textsc{Abs}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{t}_1 \rangle \Rrightarrow (C_1, \dot{\tau}_1 \to^{\nu} \dot{\tau}_2) \qquad \langle \dot{\Gamma} \vdash \dot{t}_2 \rangle \Rrightarrow (C_2, \dot{\tau}_1') \qquad \langle \dot{\tau}_1 = \dot{\tau}_1' \rangle \Rrightarrow C_3}{\langle \dot{\Gamma} \vdash \dot{t}_1\ \dot{t}_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau}_2)}\ (\textsc{App}^C)$$

$$\frac{\langle \alpha^{\nu}, \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})}{\langle \dot{\Gamma} \vdash \Lambda \alpha . \dot{t} \rangle \Rrightarrow (C, \forall^{\nu} \alpha . \dot{\tau})}\ (\textsc{TAbs}^C)$$

$$\frac{\langle \dot{\Gamma} \vdash \dot{\tau}_2 \in \mathsf{Seqable} \rangle \Rrightarrow C_1 \quad \langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C_2, \forall^{\nu} \alpha . \dot{\tau}_1) \quad \langle \dot{\tau}_1[\dot{\tau}_2/\alpha] \preceq \cdot \rangle \Rrightarrow (C_3, \dot{\tau}_3)}{\langle \dot{\Gamma} \vdash \dot{t}_{\dot{\tau}_2} \rangle \Rrightarrow (((\nu = \varepsilon) \Rightarrow C_1) \wedge C_2 \wedge C_3, \dot{\tau}_3)}\ (\textsc{TApp}^C)$$

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

$$\frac{\langle \dot{\Gamma} \vdash \dot{t}_1 \rangle \Rrightarrow (C_1, \dot{\tau}_1) \quad \langle \dot{\Gamma} \vdash \dot{\tau}_1 \in \mathsf{Seqable} \rangle \Rrightarrow C_2 \quad \langle \dot{\Gamma}, x :: \dot{\tau}_1 \vdash \dot{t}_2 \rangle \Rrightarrow (C_3, \dot{\tau}_2)}{\langle \dot{\Gamma} \vdash \textbf{let!}\ x = \dot{t}_1\ \textbf{in}\ \dot{t}_2 \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \dot{\tau}_2)}\ (\textsc{SLet}^C)$$

Figure 9: Conditional Typing Rules in **PolySeq**^{C}

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

$$\langle \dot{\Gamma} \vdash (\dot{\tau}_1 \to^{\nu} \dot{\tau}_2) \in \mathsf{Seqable} \rangle \Rrightarrow (\nu = \varepsilon)\ (\textsc{C-Arrow}^C)$$

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

$$\frac{\langle \alpha^{\varepsilon}, \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable} \rangle \Rrightarrow C}{\langle \dot{\Gamma} \vdash (\forall \alpha^{\nu} . \dot{\tau}) \in \mathsf{Seqable} \rangle \Rrightarrow C}\ (\textsc{C-TAbs}^C)$$

Figure 10: Conditional Class Membership Rules for Seqable in **PolySeq**^{C}

$$\langle \alpha \preceq \cdot \rangle \Rrightarrow (\text{True}, \alpha)\ (\textsc{S-Var}^C_1) \qquad \langle \cdot \preceq \alpha \rangle \Rrightarrow (\text{True}, \alpha)\ (\textsc{S-Var}^C_2)$$

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

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

$$\frac{\langle \dot\tau_1 \preceq \cdot \rangle \Rrightarrow (C, \dot\tau_2)}{\langle (\forall^{\nu} \alpha.\dot\tau_1) \preceq \cdot \rangle \Rrightarrow (C \wedge (\nu \leqslant \nu'), \forall^{\nu'} \alpha.\dot\tau_2)}\ (\textsc{S-All}^C_1)$$

$$\frac{\langle \cdot \preceq \dot\tau_2 \rangle \Rrightarrow (C, \dot\tau_1)}{\langle \cdot \preceq (\forall^{\nu'} \alpha.\dot\tau_2) \rangle \Rrightarrow (C \wedge (\nu \leqslant \nu'), \forall^{\nu} \alpha.\dot\tau_1)}\ (\textsc{S-All}^C_2)$$

$$\frac{\langle \dot\tau \preceq \cdot \rangle \Rrightarrow (C, \dot\tau')}{\langle [\dot\tau] \preceq \cdot \rangle \Rrightarrow (C, [\dot\tau'])}\ (\textsc{S-List}^C_1) \qquad \frac{\langle \cdot \preceq \dot\tau' \rangle \Rrightarrow (C, \dot\tau)}{\langle \cdot \preceq [\dot\tau'] \rangle \Rrightarrow (C, [\dot\tau])}\ (\textsc{S-List}^C_2)$$

Figure 11: Conditional Subtyping Rules in **PolySeq**^{C}

$$\langle \alpha = \alpha \rangle \Rrightarrow \text{True}\ (\textsc{E-Var}^C)$$

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

$$\frac{\langle \dot\tau_1 = \dot\tau_2 \rangle \Rrightarrow C}{\langle (\forall^{\nu} \alpha.\dot\tau_1) = (\forall^{\nu'} \alpha.\dot\tau_2) \rangle \Rrightarrow (C \wedge (\nu = \nu'))}\ (\textsc{E-All}^C)$$

$$\frac{\langle \dot\tau = \dot\tau' \rangle \Rrightarrow C}{\langle [\dot\tau] = [\dot\tau'] \rangle \Rrightarrow C}\ (\textsc{E-List}^C)$$

Figure 12: Conditional Equality Rules in **PolySeq**^{C}

5 Implementation

The polymorphic calculi considered so far in this paper contain only lists as algebraic data type, but the extension to other algebraic data types and base types like Int and Bool is straightforward. **PolySeq**^{C} extended by integers (with addition) and Booleans (with a case-statement) has been implemented and made usable through a web interface[2]. To facilitate more natural input, some syntactic sugar has been added. The web interface accepts closed terms according to the following grammar:

$$\begin{aligned} t ::= {} & x \mid n \mid \texttt{True} \mid \texttt{False} \mid [\,]_\tau \mid t : t \mid \textbf{case}\, t\, \textbf{of}\, \{[\,] \to t;\, x : x \to t\} \mid t + t \mid \\ & \textbf{case}\, t\, \textbf{of}\, \{\texttt{True} \to t; \texttt{False} \to t\} \mid \textbf{case}\, t\, \textbf{of}\, \{\texttt{False} \to t; \texttt{True} \to t\} \mid \\ & \textbf{if}\, t\, \textbf{then}\, t\, \textbf{else}\, t \mid \textbf{let}\, x = t\, \textbf{in}\, t \mid \lambda x :: \tau.t \mid t\, t \mid \Lambda\alpha.t \mid t_\tau \mid \\ & \textbf{fix}\, t \mid \textbf{let!}\, x = t\, \textbf{in}\, t \mid \textbf{seq}\, t\, t \end{aligned}$$

with x ranging over term variables and n ranging over the integers. Any type annotations τ in the input term must be standard type annotations (like in **PolySeq**) of the form given by the grammar $\tau ::= \alpha \mid [\tau] \mid \tau \to \tau \mid \forall\alpha.\tau \mid \mathsf{Int} \mid \mathsf{Bool}$, where α ranges over type variables. The **let**-construct is non-recursive (like **let!**), and only there to enable easy switching between presence and absence of strict evaluation. For input, the ASCII syntax is `->` for $\to$, `\` for λ, `/\` for Λ, and `forall` for $\forall$, and type subscripts are entered as `_{...}`.

The implementation returns all minimal (in the sense of minimal logical relation) refined types (with mark ε written as `e`, and $\circ$ written as `n`) the term is typable to in the refined type system. Additionally, it presents the corresponding free theorems, as well as the theorem for the standard type. The interesting parts of the theorems, namely those being related to selective strictness, are highlighted in the web interface output. A screenshot of the output for *foldl*$''$ is shown in Figure 13.

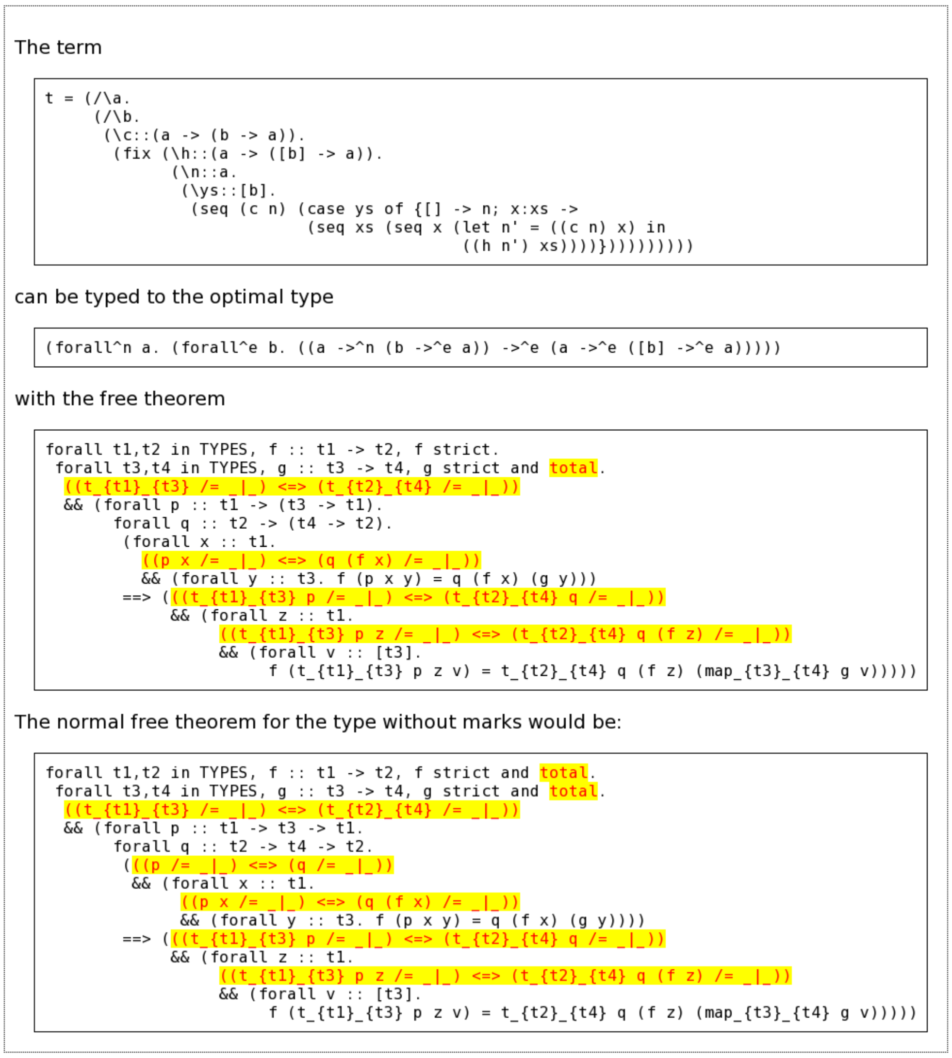
Let us comment on the respective outputs for all four initial examples, *foldl* and its strict versions *foldl*$'$, *foldl*$''$, and *foldl*$'''$. The respectively highlighted parts in the produced free theorems point out that the totality restriction on f remains required for *foldl*$'$, while the other additional restrictions mentioned in the first paragraph of Section 3 disappear. 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.[3]

6 Conclusion

The refined type system we developed and the possibility to automatically retype standardly typed terms to refined types, allow a fine-grained analysis of the influence of selective strict evaluation on free theorems, depending on its concrete use in a term. An essential difference to classical strictness analysis [Myc80, Hug86] is that we do not at-

[2] `http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/polyseq.cgi`

[3] The remaining highlighted parts in this case represent a strengthening of the theorem, not a restriction.



The term

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

can be typed to the optimal type

```
(forall^n a. (forall^e b. ((a ->^n (b ->^e a)) ->^e (a ->^e ([b] ->^e 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.
  ((t_{t1}_{t3} /= _|_) <=> (t_{t2}_{t4} /= _|_))
  && (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_{t1}_{t3} p /= _|_) <=> (t_{t2}_{t4} q /= _|_))
             && (forall z :: t1.
                  ((t_{t1}_{t3} p z /= _|_) <=> (t_{t2}_{t4} q (f z) /= _|_))
                  && (forall v :: [t3].
                       f (t_{t1}_{t3} p z v) = t_{t2}_{t4} q (f z) (map_{t3}_{t4} g v)))))
```

The normal free theorem for the type without marks 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.
  ((t_{t1}_{t3} /= _|_) <=> (t_{t2}_{t4} /= _|_))
  && (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_{t1}_{t3} p /= _|_) <=> (t_{t2}_{t4} q /= _|_))
             && (forall z :: t1.
                  ((t_{t1}_{t3} p z /= _|_) <=> (t_{t2}_{t4} q (f z) /= _|_))
                  && (forall v :: [t3].
                       f (t_{t1}_{t3} p z v) = t_{t2}_{t4} q (f z) (map_{t3}_{t4} g v)))))
```

Figure 13: Output of the Web Interface for $foldl''$ as Input

tempt to discover strict usage of arguments as resulting from "normal" execution, but only focus on explicitly enforced strict evaluation.

The calculus **PolySeq** considered, especially with the straightforward extension to algebraic data types and base types described in Section 5, is quite close to real-world lazy functional programming languages like Haskell. This permits to apply the refined free theorems for correctness proofs of transformations in such languages. For example, what is often referred to as the fusion property for $foldl$, namely the specialization of equation (1) to $g = \lambda x \to x$, is (without additional restrictions) a direct consequence of the free theorem for $foldl$'s refined type. On the other hand, the refined theorems help to easily figure out incorrect assumptions, such as that the same fusion property holds for $foldl'$.

References

[GLP93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

[Hug86] R.J.M. Hughes. Strictness detection in non-flat domains. In *Programs as Data Objects 1985, Proceedings*, volume 217 of *LNCS*, pages 112–135. Springer-Verlag, 1986.

[Joh03] P. Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13(4):797–814, 2003.

[JV04] P. Johann and J. Voigtländer. Free Theorems in the Presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

[LP96] J. Launchbury and R. Paterson. Parametricity and Unboxing with Unpointed Types. In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.

[Myc80] A. Mycroft. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *Colloque International sur la Programmation, Proceedings*, volume 83 of *LNCS*, pages 269–281. Springer-Verlag, 1980.

[SV09] D. Seidel and J. Voigtländer. Taming Selective Strictness. Technical Report TUD-FI09-06, Technische Universität Dresden, 2009. `http://wwwtcs.inf.tu-dresden.de/~voigt/TUD-FI09-06.pdf`.

[Voi08a] J. Voigtländer. Much Ado about Two: A Pearl on Parallel Prefix Computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.

[Voi08b] J. Voigtländer. Semantics and Pragmatics of New Shortcut Fusion Rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.

[Voi09a] J. Voigtländer. Bidirectionalization for Free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

[Voi09b] J. Voigtländer. Free Theorems Involving Type Constructor Classes. In *International Conference on Functional Programming, Proceedings*. ACM Press, 2009.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.