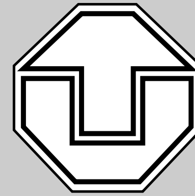
TECHNISCHE UNIVERSITÄT DRESDEN



Fakultät Informatik

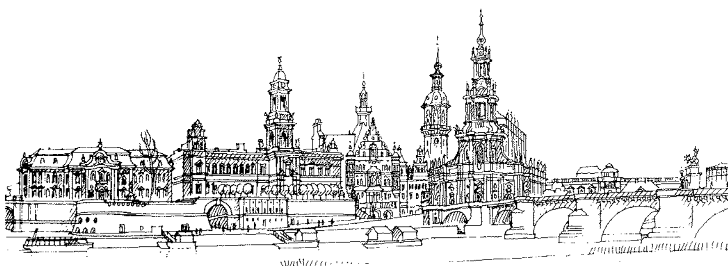
TUD-FI09-06 — Juni 2009

D. Seidel and J. Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden

Taming Selective Strictness

Technische Berichte
Technical Reports
ISSN 1430-211X



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

Taming Selective Strictness

Daniel Seidel[*] Janis Voigtländer

Institut für Theoretische Informatik
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

A Short Introduction to Free Theorems

> Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.
>
> Introduction to "Theorems for Free!" (Wadler 1989)

That is the essence of free theorems first promoted by Wadler (1989), based on Reynolds' abstraction theorem (Reynolds 1983). Formalizing the lack of type specific information of parametrically polymorphic functions, and hence their inability to do type specific transformations, Wadler develops equations that hold for each parametrically polymorphic function of a given type independently of the concrete implementation.

For example, consider the function *head* $:: \forall\alpha.[\alpha] \to \alpha$. Its type says: "For each type τ, and a list of elements of type τ as input, *head* returns some element of type τ.". Now obviously *head* has no chance to know any type specific element, for not knowing τ. It has just the structural information from the input list. Nothing more. The formalization of that fact brings us finally to the following equation:

$$head \circ (map\ f) = f \circ head \tag{1}$$

where f is an arbitrary function of type $\tau \to \tau'$, with τ and τ' arbitrary. The symbol $\circ$ is function composition, with the function on the right-hand side applied first, and *map f* applies f to each element of a list.

Although we have not mentioned the concrete action of *head* on a list, you might guess that it simply returns the first element of a list. But what if we rename *head* to *last*? Probably now we assume the function to return the last element of a list. Of course both associations are permissible and they will not harm equation (1) at all because it is independent of the concrete function *head*! Moreover one may regard the right-hand side of the equation as more simple than the left-hand side. Here we focus on a main application of free theorems: automated program transformations. Independently of the concrete functions *head* and f we can substitute the left-hand side by the right-hand side.

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

Sure, the just given example for a program transformation does not necessarily speed up a program, since a lazy language would not map f to each element of a list when afterward using just the head element, but it would do so for *last*. Disregarding that small example, there are far more important program transformations relying on free theorems. The best known is probably the *foldr*/*build*-rule (*short cut fusion*, (Gill et al. 1993)), implemented in the current GHC (Glasgow Haskell Compiler, version 6.10.1, also earlier) as on optimization rule `fold/build` removing intermediate list structures. Similar rules (*destroy*/*unfoldr*, *destroy*/*build*, ...), as well as generalized analogues to non-list algebraic data structures have also been studied (Svenningsson 2002; Johann 2002; Voigtländer 2008).

The original investigation of free theorems was made in the pure polymorphic lambda calculus (Girard/Reynolds calculus, System F) which is, in particular due to the inability to capture general recursion, a radical simplification compared to real-world functional programming languages. The restrictions arising with the introduction of a fixpoint operator, not definable in a pure polymorphic calculus, have been known from the beginning (Wadler 1989), and their necessity has already been studied (Launchbury and Paterson 1996; Seidel and Voigtländer 2009). In general, the presence of an undefined value (denoted as $\bot$), as imposed by diverging recursions and incomplete pattern matches, forces strictness conditions on free theorems. For example, f in equation (1) has to be strict (i.e. $f\ \bot = \bot$) to guarantee the equation when an undefined value is present[1]. Strictness conditions are not the focus in this paper, and we just take them as required from now on. Our interest is in the influence of selective strict evaluation on free theorems.

Trouble with Selective Strictness Modern lazy functional programming languages like Haskell and Clean extend the pure polymorphic lambda calculus not only by a fixpoint combinator, they additionally allow selective strictness. In Haskell strict evaluation is provided through the function *seq*, taking two arguments and forcing the first to be evaluated before returning the second one. Note that this is not a deep *seq*, i.e. it only returns undefined if its first argument is completely undefined. For nested undefined values like a list with undefined elements, it returns its second argument. Similar features are strict data constructors and the strict function application "\$!" forcing the function's arguments to be evaluated even if not used. Although *seq* was primarily introduced into Haskell to improve speed (Hudak et al. 2007), the big advantage of selective strictness is the possibility to remove space leaks otherwise likely to arise in call-by-need languages. The disadvantage lies in the weakening of parametricity, and hence the free theorems built on it. This, for example, is especially distressing because the already mentioned optimization rules like short-cut fusion, used in the Haskell compiler GHC, rely on parametricity and hence may become unsound.[2]

To get an impression of the problems that arise by the introduction of selective strictness, we regard the well-known Haskell prelude function *foldl*, its strict double *foldl′* (in the Haskell module `Data.List`), as well as the functions *foldl″* and *foldl‴* which force strict evaluation at rather arbitrary points. Possible implementations in Haskell are shown in Figure 1. Strict evaluation is captured via *seq*.

All four functions are of type $\forall a.\forall b.(a \to b \to a) \to a \to [b] \to a$, and the corresponding free theorem, disregarding strict evaluation, states

$$f\ (foldl\ c\ n\ xs) = foldl\ c'\ (f\ n)\ (map\ g\ xs) \tag{2}$$

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

As an example how easy we prove well-known results by free theorems, taking g as the identity function ($id = \lambda x \to x$), we get the fusion property (Hutton 1999) of *foldl* and hence, it can be proved directly by the free theorem in the absence of selective strictness. When we take strict

[1] And indeed, it has to be present for functions of that type to handle the case with an empty list as input.

[2] The expression `foldr undefined 0 (build seq)` will return `0` when the program is compiled without optimization and `Prelude.undefined` when compiled with the option `-O`, because then the `fold/build`-rule fires. The function `build` can be found in the module `GHC.Exts`.

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

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

Figure 1: Variations of the *foldl* Function with Different Uses of *seq*.

evaluation into account, the situation changes. For the Haskell function *foldl′* we can show that the free theorem and also the fusion property do not hold.

Consider equation (2) with the instantiations

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

Regarding *foldl* everything is fine, but for the strict *foldl′* we get $True = \bot$. In that case it suffices to restrict f to be total, but if we consider the functions *foldl′′* and *foldl′′′*, for which the just given instantiation does not break the free theorem, we will detect the necessity of further restrictions. Consider each of the following instantiations respectively:

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

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 the instantiations equation (2) holds for *foldl* and *foldl′*, but the first and the second instantiation fail for *foldl′′*, while the last one fails for *foldl′′′*. All three failures go back to different uses of *seq*, forcing different restrictions. The use of *seq* on c forces that $c \neq \bot$ iff $c' \neq \bot$, *seq* on $(c\ n)$ forces that $c\ z \neq \bot$ iff $c'\ (f\ z) \neq \bot$ for all z and *seq* on x forces g to be total. Only the strict evaluation of the list xs requires no additional condition. This might be apparent, since strict evaluation of lists can be forced also by a simple case statement.

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 consequence is the question how we can express detailed information about the use of strict evaluation such that it can influence the restrictions on free theorems. Since free theorems only depend on the type of a term, the information has to be part of the type signature, and that is exactly our approach.

Taming Selective Strictness The idea is to keep track of strict evaluation in the type signatures. This is not a new proposal and the first implementation of Haskell which supported selective strictness[3] treated *seq* as an overloaded function, rather than a fully polymorphic one, making its use visible in the type signature through the type class `Eval`. Because of the necessity to change

[3] Haskell version 1.3

lots of type signatures when inserting a single *seq*, the idea was discarded in the later Haskell 98 standard, making *seq* fully polymorphic (Hudak et al. 2007).

But, even when present, the type class approach was not capable to capture all effects of strict evaluation since it presumes that the effects can be read off from marks at type variables only. This is not given. For example $foldl'''$ in Figure 1 would have no `Eval`-restriction at all, but clearly, as just stated by an example, the use of *seq* on c causes problems. Indeed, the Haskell report version 1.3 (Section 6.2.7)writes "Functions as well as all other built-in types are in `Eval`.". But even if we consider function types to be in general not in `Eval`, and force their membership explicitly by allowing type class restrictions on function types[4], the problems of the approach remain. Consider a function $f :: \texttt{Eval}(\alpha \to \mathsf{Int}) \Rightarrow (\alpha \to \mathsf{Int}) \to (\alpha \to \mathsf{Int}) \to \mathsf{Int}$. It could be of the form $f = \lambda g\ h \to \ldots$ where `Eval` restricts g and h to be suitable for strict evaluation. But what if we need the restriction only for g? From the type class approach, there is no way to tell that *seq* is only used on one of the functions.

To avoid these problems, we choose another way to track strict evaluation in the type. We provide special annotations at quantification of type variables and also at function types. This leads to a clear correspondence to the impact of strict evaluation on free theorems. Combining the insights of Johann and Voigtländer (2004) with ideas of Launchbury and Paterson (1996) regarding general recursion, we present a calculus allowing for refined free theorems by a refined type system. We then develop an algorithm computing all refined types for a given term.

Our testbed is a standard lambda calculus with a fixpoint and a strictness primitive, very similar to Haskell, but with explicit type annotations. We refine the calculus by adding a second version of type constructors for function types and quantification of type variables, adjust the typing rules, and prove the parametricity theorem for the refined calculus, which allows more control over the restrictions on free theorems. Afterwards, we provide an algorithm gaining all possible refined types for a given ordinarily typed term. This allows the automatized derivation of strong free theorems for a term typed in the standard calculus. In particular, we are able to clarify automatically which *seq* in $foldl'$, $foldl''$, and $foldl'''$ in Figure 1 causes which restriction.

Structure of the Paper After we have introduced the standard calculus **PolySeq** in Section 2 and recalled the parametricity theorem for that calculus, we move on to a calculus with a refined type system, called **PolySeq***, in Section 3. Here we prove equivalence of expressiveness to the standard calculus. Again we give the, now stronger, parametricity theorem. Section 4 presents a further alteration of **PolySeq*** with the same set of typable terms, but with improved algorithmic properties. The last calculus, $\mathbf{PolySeq}^C$, is presented in Section 5. It is again equivalent in expressiveness to the previous calculi, but acts on parameterized types, allowing a term to be attached to the set of all its possible types. The typing rules, still written in a declarative style, can be used algorithmically to gain all possible refined types for a given term with standard type annotations. Up to that point, we abstain from base types and algebraic types other than lists in our calculi. Section 6 closes that gap by indicating the straightforward nature of these extensions. Moreover, it points to an implementation of $\mathbf{PolySeq}^C$, ready to use online[5], and not only assigning refined types to a term, but also showing refined free theorems for the obtained types.

2 Standard Parametricity

We start out from a standard denotational semantics for a polymorphic lambda-calculus that corresponds to Haskell.

The syntax of types and terms is given in Figure 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

[4] This is not allowed in Haskell 98, but as an extension in GHC, enabled by `-XFlexibleContexts`.
[5] `http://linux.tcs.inf.tu-dresden.de/~seideld/cgi-bin/polyseq.cgi`

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

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

Figure 2: Syntax of Types τ and Terms t.

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

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

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

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

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

Figure 3: Typing Rules in **PolySeq** (and later **PolySeq***), Part 1.

well. General recursion is captured via a fixpoint primitive, while selective strictness (à la *seq*) is provided via a strict-let construct.

Figures 3 and 4 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 $\vdash \mathit{map} :: \tau$, where $\tau = \forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$:

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

The denotational semantics interprets types as *pointed complete partial orders* (for short, *pcpos*; least element always denoted $\bot$). The definition in Figure 5, 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 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 5, 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 6 is also standard. It uses λ for denoting anonymous functions, and the following operator:

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

The expression $\bigsqcup_{n \geq 0}\ ([\![t]\!]_{\theta,\sigma}\ \$)^n\ \bot$ in the definition for **fix** means the supremum of the chain

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

Figure 4: Typing Rules in **PolySeq** , Part 2.

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

Figure 5: Semantics of Types.

$$
\begin{aligned}
[\![x]\!]_{\theta,\sigma} &= \sigma(x) \\
[\![[\,]_\tau]\!]_{\theta,\sigma} &= \lfloor [\,] \rfloor \\
[\![t_1 : t_2]\!]_{\theta,\sigma} &= \lfloor [\![t_1]\!]_{\theta,\sigma} : [\![t_2]\!]_{\theta,\sigma} \rfloor \\
[\![\textbf{case } t \textbf{ of } \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma} &= \\
&\begin{cases} [\![t_1]\!]_{\theta,\sigma} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor [\,] \rfloor \\ [\![t_2]\!]_{\theta,\sigma[x_1\mapsto a,\, x_2\mapsto b]} & \text{if } [\![t]\!]_{\theta,\sigma} = \lfloor a : b \rfloor \\ \perp & \text{if } [\![t]\!]_{\theta,\sigma} = \perp \end{cases} \\
[\![\lambda x :: \tau.t]\!]_{\theta,\sigma} &= \lfloor \lambda a.[\![t]\!]_{\theta,\sigma[x\mapsto a]} \rfloor \\
[\![t_1\ t_2]\!]_{\theta,\sigma} &= [\![t_1]\!]_{\theta,\sigma}\ \$\ [\![t_2]\!]_{\theta,\sigma} \\
[\![\Lambda\alpha.t]\!]_{\theta,\sigma} &= \lambda D.[\![t]\!]_{\theta[\alpha\mapsto D],\sigma} \\
[\![t_\tau]\!]_{\theta,\sigma} &= [\![t]\!]_{\theta,\sigma}\ [\![\tau]\!]_\theta \\
[\![\textbf{fix } t]\!]_{\theta,\sigma} &= \textstyle\bigsqcup_{n\geq 0}\ ([\![t]\!]_{\theta,\sigma}\ \$)^n\ \perp \\
[\![\textbf{let!}\ x = t_1 \textbf{ in } t_2]\!]_{\theta,\sigma} &= \begin{cases} [\![t_2]\!]_{\theta,\sigma[x\mapsto a]} & \text{if } [\![t_1]\!]_{\theta,\sigma} = a \neq \perp \\ \perp & \text{if } [\![t_1]\!]_{\theta,\sigma} = \perp \end{cases}
\end{aligned}
$$

Figure 6: Semantics of Terms.

$\perp \sqsubseteq ([\![t]\!]_{\theta,\sigma}\ \$\ \perp) \sqsubseteq ([\![t]\!]_{\theta,\sigma}\ \$\,([\![t]\!]_{\theta,\sigma}\ \$\ \perp)) \cdots$. 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 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, $\mathit{Rel}(D_1, D_2)$ 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 $(\perp,\perp)$ and closure under suprema. A relation $\mathcal{R}$ is bottom-reflecting if $(a,b) \in \mathcal{R}$ implies that $a = \perp$ iff $b = \perp$. 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. 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 $\Delta_{\tau,\rho} \in \mathit{Rel}$ as well.*

The lemma is crucial for then proving the following parametricity theorem.

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

Figure 7: Standard Logical Relation.

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

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

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

Figure 8: Class Membership Rules for Seqable in **PolySeq*** (and later **PolySeq$^+$**).

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, $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$ 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 can be found in Appendix A.

3 Refining the Calculus

If we recall the fold functions from the introduction and the "naive" version of the corresponding free theorem, stated in equation (2), we can compare that version with the "secure" version arising from Theorem 2.2. The "secure" theorem requires f and g total, $c = \bot$ iff $c' = \bot$ and $c\ z = \bot$ iff $c'\ (f\ z) = \bot$ for all z in addition to the restrictions on the "naive" theorem.

As already seen in the introduction and also visible from the proof of Theorem 2.2, these additional restrictions arise from different uses of strict evaluation and are each respectively only necessary if strict evaluation is forced at a particular place. Hence, it is reasonable to make strict evaluation (and the place of its use) visible from the type of a term. If we can predict by the type where strict evaluation, i.e. (SLet), is used, or better where not, we could, depending on that information, release some of the restrictions newly arising in the "secure" free theorem. The idea is to extend the set of type constructors to distinguish types that can be used for strictly evaluated terms from types that cannot.

First, we distinguish two kinds of type variables, one allowed to be used as type of strictly evaluated terms, and one not. The first is marked by ε in the typing environment and quantified by $\forall^{\varepsilon}$, while the second is marked by $\circ$ when appearing in the typing environment and quantified by $\forall^{\circ}$. Furthermore, for function types we take $\to^{\varepsilon}$ as a constructor that produces types suitable for strictly evaluated terms and introduce a second constructor $\to^{\circ}$ which produces therefor unsuitable types. In the case of lists, strict evaluation is uninteresting because it can easily be imitated by a case statement and hence is no extension compared to a calculus without strict evaluation.

We can summarize the types whose terms are suitable to be strictly evaluated in a class Seqable that is defined by the rules shown in Figure 8. The first three rules correspond directly to the just given information. The rules $(\text{C-TAbs}_{\nu})_{\nu \in \{\circ,\varepsilon\}}$, parameterized by ν, arise because we do not distinguish between $\bot$ and the polymorphic value that is $\bot$ for all type instantiations (since Haskell does not either). By taking α^{ε} instead of α^{ν} in the premise, we in excess put the type $\forall^{\circ}\alpha.\alpha$ into Seqable[6]. The single inhabitant of that type is $\bot$, which leads to **let!** $x = t_1$ **in** t being equal to $\bot$ and hence replaceable by **fix** $(\lambda x :: \tau.x)$ for appropriate τ. Therefore that case is not of interest.

With the system just defined we can restrict the use of (SLet) introducing strict evaluation such that τ_1 in the rule has to be in Seqable. Also for type applications we allow only Seqable types as instances for type variables quantified by $\forall^{\varepsilon}$. Additionally, a rule for type application to terms of type $\forall^{\circ}\alpha.\tau$ is needed. These rules are shown in Figure 9 and replace the ones from Figure 4.

[6]More exactly, we also include $\forall^{\circ}\alpha.\forall^{\nu_1}\beta_1.\dots.\forall^{\nu_n}\beta_n.\alpha$ in Seqable, but these types are as well only inhabited by $\bot$.

$$\frac{\Gamma \vdash \tau_2 \in \mathsf{Seqable} \qquad \Gamma \vdash t :: \forall\alpha.\tau_1}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\ (\textsc{TApp'}) \qquad \frac{\Gamma \vdash t :: \forall^{\circ}\alpha.\tau_1}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\ (\textsc{TApp'}_{\circ})$$

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

Figure 9: Replacements for the Rules in Figure 4 in **PolySeq***.

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

$$\frac{\alpha^{\circ}, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda\alpha.t) :: \forall^{\circ}\alpha.\tau}\ (\textsc{TAbs}_{\circ}) \qquad \frac{\Gamma \vdash t :: \tau \to^{\circ} \tau}{\Gamma \vdash \textbf{fix}\ t :: \tau}\ (\textsc{Fix}_{\circ})$$

$$\frac{\Gamma \vdash t :: \tau_1 \qquad \tau_1 \preceq \tau_2}{\Gamma \vdash t :: \tau_2}\ (\textsc{Sub})$$

Figure 10: New Typing Rules in **PolySeq***.

Notice that we apply the convention that the mark ε is the "invisible" mark and can be dropped. This is used throughout the rest of the paper.

The rules of Figure 3 remain part of the typing rules. But the new $\circ$-marked type constructors lead to additional rules, shown in Figure 10. All extensions are straightforward. Only the rule (Sub) requires some explanation. It is motivated by 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 $\lambda f :: \tau_1 \to \tau_2.\,\textbf{let!}\ x = f\ \textbf{in}\ []$ is only typable to the second type. So whether strict evaluation is allowed at arguments of a function or not decides over the subtype relation. The subtyping system presented in Figure 11 goes a bit further and restricts the relation thus that a Seqable supertype forces a Seqable subtype. We can think of that as follows: the set of functions that need to be allowed to be strictly evaluated is only part of the set of functions that do not. The rules are written as parameterized rule families, where $\{\circ, \varepsilon\}$ is the ordered set of marks with $\circ < \varepsilon$.

With the extended type system and the appropriately extended typing rules a refined calculus is given. We call it **PolySeq***. Before moving on to the semantics, we state the syntactic correspondence to **PolySeq**. The following definition helps to abstract from the marks at type variables, types, and type annotations.

Definition 3.1 (mark eraser). The function $|\cdot|$ takes a term, type, or typing environment in **PolySeq*** and returns it with all marks at type variables, $\forall$-quantifiers, and arrows removed. The result is called the *erasure* of the input.

With the help of Definition 3.1 we can express that the set of typable terms in **PolySeq** and **PolySeq*** is equivalent up to the marks at the type annotations.

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

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

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

Figure 11: Subtyping Rules for **PolySeq*** (and **PolySeq^{+}**).

Lemma 3.2. *If we have Γ, t, τ such that $\Gamma \vdash t :: \tau$ in* ***PolySeq****, then $\Gamma \vdash t :: \tau$ holds in* ***PolySeq*****.* *If we have Γ, t, τ such that $\Gamma \vdash t :: \tau$ in* ***PolySeq*****, then $|\Gamma| \vdash |t| :: |\tau|$ holds in* ***PolySeq****.*

For the proof of Lemma 3.2 two auxiliary lemmas are necessary.

Lemma 3.3. *If a type τ is closed under a typing environment Γ then $|\Gamma| \vdash |\tau| \in \mathsf{Seqable}$ holds.*

Proof. Easy induction over the type structure of τ.

Lemma 3.4. *Let τ, τ' be two types. If $\tau \preceq \tau'$ then $|\tau| = |\tau'|$.*

Proof. Induction over the derivation tree of $\tau \preceq \tau'$. Obvious from the rules of Figure 11.

Proof (of Lemma 3.2). For the first part of the lemma, assume we have t typable to τ under Γ in **PolySeq**. Then there exists a derivation tree $\mathcal{T}$ leading to $\Gamma \vdash t :: \tau$ in **PolySeq**. We show that there exists a derivation tree $\mathcal{T}'$ in **PolySeq*** leading to $\Gamma \vdash t :: \tau$ as well.

The proof that such a $\mathcal{T}'$ exists is done inductively over the depth of the derivation tree $\mathcal{T}$. Thus we regard only the rule at the root of $\mathcal{T}$ and translate it into a rule of **PolySeq***.

All rules from Figure 3 stay unchanged. Hence, it remains to consider the rules of Figure 4, namely (TApp) and (SLet). They can be replaced by (TApp') and (SLet') from Figure 9, respectively. The additional premises are fulfilled by Lemma 3.3.

Now, regard the second part of the lemma. Assume, we have given a derivation tree $\mathcal{T}'$ in **PolySeq***, leading to $\Gamma \vdash t :: \tau$. We prove by induction on the depth of $\mathcal{T}'$, regarding only the root rule, that there exists a derivation tree $\mathcal{T}$ in **PolySeq**, leading to $|\Gamma| \vdash |t| :: |\tau|$. Obviously, each rule from Figure 3 can remain and the $\circ$-marked versions of these rules from Figure 10 can be substituted by the unmarked ones. Also (SLet') can be replaced by (SLet), and (TApp') as well as (TApp'$_\circ$) by (TApp).

It remains a translation of (Sub). This rule can just be skipped by Lemma 3.4.

The next step is to consider the semantics of **PolySeq***. Our aim is to take over the semantics of **PolySeq** and we easily do it the following way:

Definition 3.5. Let $\Gamma \vdash t :: \tau$ in **PolySeq*** and θ a mapping from all type variables in Γ to pcpos. The semantics of τ is defined by $[\![|\tau|]\!]_\theta$. Furthermore, with $\sigma(x) \in [\![|\tau'|]\!]_\theta$ for all $x :: \tau'$ in Γ, the semantics of t is $[\![|t|]\!]_{\theta,\sigma}$.

Having ensured the new system **PolySeq*** is equivalent to **PolySeq** in terms of typability and semantics, and thus having it proved to be suitable as a refined system, we go on considering parametricity, and state and prove the (refined) parametricity theorem for **PolySeq***.

The typing rules ensure that terms of types $\tau_1 \to^\circ \tau_2$ and α, if α is marked by $\circ$ in the corresponding typing environment or quantified by $\forall^\circ$, are not strictly evaluated, i.e. these types do not appear as type τ_1 in the rule (SLet'). Recalling the proof of Theorem 2.2 from the appendix, we notice that only for τ_1 in (SLet') the logical relation has to be bottom-reflecting. Hence, for the definition of the relational actions for $\forall^\circ\alpha.\tau$ and $\tau_1 \to^\circ \tau_2$ we can omit the bottom-reflection restrictions and define them as

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

where $Rel^\circ(D_1,D_2)$ 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 7).

Before we state the refined parametricity theorem, we have to formally prove that the logical relation just defined is really bottom-reflecting for all types in $\mathsf{Seqable}$ and also that it is strict and continuous for every type. Furthermore, the meaning of subtyping to the logical relation has to be clarified. It turns out that a subtype relation between two types has a very natural interpretation

as the logical relation of the subtype being a subset of the logical relation of the supertype. The next two lemmas state these properties.

We call a mapping ρ *appropriate* to a typing environment Γ if the type variables in Γ are the domain of ρ and $\rho(\alpha) \in \mathit{Rel}$ for each $\alpha^{\varepsilon} \in \Gamma$, as well as $\rho(\alpha) \in \mathit{Rel}^{\circ}$ for each $\alpha^{\circ} \in \Gamma$, where Rel° is the union of all $\mathit{Rel}^{\circ}(D_1, D_2)$.

Lemma 3.6. *Let τ be a type closed under the typing environment Γ. Then*

1. $\Delta_{\tau,\rho} \in \mathit{Rel}^{\circ}$ *and*
2. $\Gamma \vdash \tau \in \mathsf{Seqable} \Rightarrow \Delta_{\tau,\rho} \in \mathit{Rel}$

for each appropriate ρ.

Proof. The first statement of the lemma holds by induction on the type structure of τ, using the relational actions defining the logical relation.

Regarding the second statement we do an induction over the rules of the Seqable-system (cf. Figure 8). For the axioms (C-List), (C-Arrow) and (C-Var) the lemma is immediate from the definition of the logical relation.

In the case (C-TAbs$_{\circ}$), we have the following proof:

$$
\begin{array}{cl}
 & \Delta_{\forall^{\circ}\alpha.\tau,\rho} \in \mathit{Rel} \\
\Leftrightarrow & \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^{\circ}(D_1,D_2).\ (u\ D_1, v\ D_2) \in \Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\} \in \mathit{Rel} \\
\Leftarrow & (\forall (u,v) \in \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^{\circ}(D_1,D_2).\ (u\ D_1, v\ D_2) \in \Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\} \\
 & \Rightarrow (u = \bot \Leftrightarrow v = \bot)) \\
\Leftarrow & \forall D_1, D_2 \text{ pcpos}.\ \exists \mathcal{R} \in \mathit{Rel}^{\circ}.\ (\Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]} \in \mathit{Rel}) \\
\Leftarrow & \forall D_1, D_2 \text{ pcpos}.\ \exists \mathcal{R} \in \mathit{Rel}.\ (\Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]} \in \mathit{Rel}) \\
\Leftarrow & \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}(D_1,D_2).\ (\Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]} \in \mathit{Rel}) \\
\Leftarrow & \alpha, \Gamma \vdash \tau \in \mathsf{Seqable}
\end{array}
$$

For (C-TAbs) we skip the fourth line of the reasoning and replace Rel° by Rel in the second and third line, as well as $\forall^{\circ}$ by $\forall$ in the first line.

Lemma 3.7 (Subtyping). *If $\tau_1 \preceq \tau_2$, then $\Delta_{\tau_1,\rho} \subseteq \Delta_{\tau_2,\rho}$.*

Proof. We prove inductively over the subtyping derivation and thus regard each rule in Figure 11 separately, assuming the premises as induction hypotheses and taking ρ as arbitrary, but such that the type variables occurring freely in the inspected types form the domain of ρ.

For (S-Var) the result is immediate.

Regarding the rules (S-Arrow$_{\nu,\nu'}$), we first consider $\nu = \nu' = \circ$. By

$$
\begin{array}{cl}
 & \Delta_{\sigma_1 \to^{\circ} \sigma_2,\rho} \subseteq \Delta_{\tau_1 \to^{\circ} \tau_2,\rho} \\
\Leftrightarrow & \{(f,g) \mid \forall (a,b) \in \Delta_{\sigma_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\sigma_2,\rho}\} \\
 & \subseteq \{(f,g) \mid \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau_2,\rho}\} \\
\Leftarrow & \Delta_{\tau_1,\rho} \subseteq \Delta_{\sigma_1,\rho} \wedge \Delta_{\sigma_2,\rho} \subseteq \Delta_{\tau_2,\rho} \\
\Leftarrow & \tau_1 \preceq \sigma_1 \wedge \sigma_2 \preceq \tau_2
\end{array}
$$

the induction hypotheses suffice. The same reasoning works for $\nu = \nu' = \varepsilon$. The last case with $\nu = \varepsilon$ and $\nu' = \circ$ is direct by the previous two and $\Delta_{\tau \to \tau',\rho} \subseteq \Delta_{\tau \to^{\circ} \tau',\rho}$, which is from the definition of the logical relation.

For (S-All$_{\nu,\nu'}$) we have a similar reasoning with the three distinct cases, using the definition of the logical relation in each case. We consider only $\nu = \nu' = \circ$, for $\nu = \nu' = \varepsilon$ is similar and the last case is direct from the first two with $\Delta_{\forall^{\circ}\alpha.\tau,\rho} \subseteq \Delta_{\forall\alpha.\tau,\rho}$. For $\nu = \nu' = \circ$ we have

$$
\begin{array}{cl}
 & \Delta_{\forall^{\circ}\alpha.\tau_1,\rho} \subseteq \Delta_{\forall^{\circ}\alpha.\tau_2,\rho} \\
\Leftrightarrow & \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^{\circ}(D_1,D_2).\ (u\ D_1, v\ D_2) \in \Delta_{\tau_1,\rho[\alpha\mapsto\mathcal{R}]}\} \\
 & \subseteq \{(u,v) \mid \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^{\circ}(D_1,D_2).\ (u\ D_1, v\ D_2) \in \Delta_{\tau_2,\rho[\alpha\mapsto\mathcal{R}]}\} \\
\Leftarrow & \forall \mathcal{R} \in \mathit{Rel}^{\circ}.\ (\Delta_{\tau_1,\rho[\alpha\to\mathcal{R}]} \subseteq \Delta_{\tau_2,\rho[\alpha\to\mathcal{R}]}) \\
\Leftarrow & \tau_1 \preceq \tau_2\,.
\end{array}
$$

Finally, the case (S-List) is by

$$\begin{array}{ll} & \Delta_{[\sigma],\rho} \subseteq \Delta_{[\tau],\rho} \\ \Leftrightarrow & \mathit{list}\,\Delta_{\sigma,\rho} \subseteq \mathit{list}\,\Delta_{\tau,\rho} \\ \Leftarrow & \Delta_{\sigma,\rho} \subseteq \Delta_{\tau,\rho} \\ \Leftarrow & \sigma \preceq \tau\,. \end{array}$$

The following parametricity theorem is stronger than the one in **PolySeq**. for example, by using ∘-marks we can type a term t to a type τ under an environment Γ, with $\Delta_{\tau,\rho}$ in **PolySeq*** possibly a subset of the logical relation for unmarked, but elsewise syntactically equivalent, Γ, t, and τ in **PolySeq**.

Theorem 3.8 (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 $([\![|t|]\!]_{\theta_1,\sigma_1}, [\![|t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho}$.

Proof. The proof is very similar to the one of Theorem 2.2, given in the appendix. We concentrate only on the interesting, differing cases.

In the case (Cons) we use Lemma 3.6(1) instead of Lemma 2.1. Regarding (Abs_ν) for $\nu \in \{\circ, \varepsilon\}$, the reasoning is, independently of ν, similar to the one for (Abs) in the proof of Theorem 2.2. The same holds for (App_ν) and (App).

In the cases

$$\frac{\alpha^\nu, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda\alpha.t) :: \forall^\nu\alpha.\tau}\ (\text{TAbs}_\nu)_{\nu\in\{\circ,\varepsilon\}}\,,$$

for $\nu = \circ$ and $\nu = \varepsilon$ we have

$$\begin{array}{ll} & ([\![|\Lambda\alpha.t|]\!]_{\theta_1,\sigma_1}, [\![|\Lambda\alpha.t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\forall^\nu\alpha.\tau,\rho} \\ \Leftrightarrow & (\lambda D_1.[\![|t|]\!]_{\theta_1[\alpha\mapsto D_1],\sigma_1}, \lambda D_2.[\![|t|]\!]_{\theta_2[\alpha\mapsto D_2],\sigma_2}) \in \Delta_{\forall^\nu\alpha.\tau,\rho} \\ \Leftrightarrow & \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^\nu(D_1, D_2). \\ & \quad ([\![|t|]\!]_{\theta_1[\alpha\mapsto D_1],\sigma_1}, [\![|t|]\!]_{\theta_2[\alpha\mapsto D_2],\sigma_2}) \in \Delta_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\,, \end{array}$$

where $\mathit{Rel}^\varepsilon = \mathit{Rel}$ by convention.

In the cases

$$\frac{(\text{if } \nu = \varepsilon \text{ then } \Gamma \vdash \tau_2 \in \mathsf{Seqable}) \qquad \Gamma \vdash t :: \forall^\nu\alpha.\tau_1}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\ (\text{TApp'}_\nu)_{\nu\in\{\circ,\varepsilon\}}\,,$$

we have

$$\begin{array}{ll} & ([\![|t_{\tau_2}|]\!]_{\theta_1,\sigma_1}, [\![|t_{\tau_2}|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1[\tau_2/\alpha],\rho} \\ \Leftrightarrow & ([\![|t|]\!]_{\theta_1,\sigma_1}\ [\![|\tau_2|]\!]_{\theta_1}, [\![|t|]\!]_{\theta_2,\sigma_2}\ [\![|\tau_2|]\!]_{\theta_2}) \in \Delta_{\tau_1,\rho[\alpha\mapsto\Delta_{\tau_2,\rho}]} \\ \Leftarrow & \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}^\nu(D_1, D_2). \\ & \quad ([\![|t|]\!]_{\theta_1,\sigma_1}\ D_1, [\![|t|]\!]_{\theta_2,\sigma_2}\ D_2) \in \Delta_{\tau_1,\rho[\alpha\mapsto\mathcal{R}]} \\ \Leftrightarrow & ([\![|t|]\!]_{\theta_1,\sigma_1}, [\![|t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\forall^\nu\alpha.\tau_1,\rho}\,, \end{array}$$

independently of ν, and so the induction hypothesis suffices.

Note that the equivalence $\Delta_{\tau_1[\tau_2/\alpha],\rho} = \Delta_{\tau_1,\rho[\alpha\mapsto\Delta_{\tau_2,\rho}]}$, used in the first step, holds by an easy induction on τ_1. Also note that the consecutive step uses $\Delta_{\tau_2,\rho} \in \mathit{Rel}^\circ$ (for $\nu = \circ$), as justified by Lemma 3.6(1), and $\Delta_{\tau_2,\rho} \in \mathit{Rel}$ for $\nu = \varepsilon$, as justified by the additional premise in (TApp'_ε) and Lemma 3.6(2).

For (Fix_ν), $\nu \in \{\circ, \varepsilon\}$, the reasoning is similar to the one for (Fix) in the proof of Theorem 2.2.

In the case

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

we reason by the set inclusion $\Delta_{\tau_1,\rho} \subseteq \Delta_{\tau_2,\rho}$, which is true as shown in Lemma 3.7:

$$\begin{array}{ll} & ([\![|t|]\!]_{\theta_1,\sigma_1}, [\![|t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_2,\rho} \\ \Leftarrow & ([\![|t|]\!]_{\theta_1,\sigma_1}, [\![|t|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}\,. \end{array}$$

Finally, in the case

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

we have to show that the values

$$\begin{cases} [\![|t_2|]\!]_{\theta_1,\sigma_1[x\mapsto a]} & \text{if } [\![|t_1|]\!]_{\theta_1,\sigma_1} = a \neq \bot \\ \bot & \text{if } [\![|t_1|]\!]_{\theta_1,\sigma_1} = \bot \end{cases}$$

and

$$\begin{cases} [\![|t_2|]\!]_{\theta_2,\sigma_2[x\mapsto b]} & \text{if } [\![|t_1|]\!]_{\theta_2,\sigma_2} = b \neq \bot \\ \bot & \text{if } [\![|t_1|]\!]_{\theta_2,\sigma_2} = \bot \end{cases}$$

are related by $\Delta_{\tau_2,\rho}$. By the induction hypothesis $([\![|t_1|]\!]_{\theta_1,\sigma_1}, [\![|t_1|]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}$ and bottom-reflection of $\Delta_{\tau_1,\rho}$, obtained by the premise $\Gamma \vdash \tau_1 \in \mathsf{Seqable}$ and Lemma 3.6(2), it suffices to consider the following two cases:

1. $[\![|t_1|]\!]_{\theta_1,\sigma_1} = a \neq \bot$ and $[\![|t_1|]\!]_{\theta_2,\sigma_2} = b \neq \bot$, in which case the induction hypothesis that for every $(a, b) \in \Delta_{\tau_1,\rho}$,
$$([\![|t_2|]\!]_{\theta_1,\sigma_1[x\mapsto a]}, [\![|t_2|]\!]_{\theta_2,\sigma_2[x\mapsto b]}) \in \Delta_{\tau_2,\rho}\,,$$
suffices, and
2. $[\![|t_1|]\!]_{\theta_1,\sigma_1} = \bot$ and $[\![|t_1|]\!]_{\theta_2,\sigma_2} = \bot$, in which case we have to show $(\bot, \bot) \in \Delta_{\tau_2,\rho}$, which follows from the strictness of $\Delta_{\tau_2,\rho}$ (cf. Lemma 3.6(1)).

This completes the proof.

Let us finish this section by an example for a refined type and the corresponding free theorem. Recall the function *foldl''* from the introduction. It can be typed to $\forall^\circ a.\forall b.(a \to^\circ b \to a) \to a \to [b] \to a$ in **PolySeq***, and the corresponding free theorem states equation (2) if, additionally to the conditions in the introduction, g is total and $c\ z = \bot$ iff $c'\ (f\ z) = \bot$ holds for all z. Compared to the "secure" free theorem from the beginning of this section we do not request f to be total and $c = \bot$ iff $c' = \bot$ anymore. Hence we get rid of unnecessary restrictions, and gain a stronger and still "secure" free theorem for *foldl''*.

4 Removing the (Sub)-Rule

The calculus **PolySeq*** with the refined type system, presented in the previous section, enables us to release some restrictions on free theorems, if the use of strict evaluation is localized by the type. Thus it allows for stronger free theorems compared to the standard calculus **PolySeq**. Our further goal is to automatically type terms with standard type annotations like in **PolySeq** to a refined type of **PolySeq***. But, unfortunately, **PolySeq*** is not very suitable for a use as typing algorithm. The reason is the (Sub) rule, being in competition with all other rules. Hence, we are looking for a calculus with the same type system and the same terms typable as in the just presented **PolySeq***, but without the rule (Sub).

The idea is to look at the type derivation trees constructible by the typing rules of **PolySeq*** and consider where subtyping is really necessary and where it can be shifted to another place in the derivation tree. If an application of (Sub) is forced to be done *after* some rule, we manipulate that rule to integrate the transformation of (Sub) directly.

The resulting calculus has for each typing rule of **PolySeq***, beside (Sub), a corresponding rule. The subtyping and the Seqable rule system remain unchanged. We call the new calculus

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

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

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

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

$$\frac{\alpha^\nu, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda \alpha.t) :: \forall^\nu \alpha.\tau} \; (\textsc{TAbs}_\nu)_{\nu \in \{\circ, \varepsilon\}}$$

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

$$\frac{(\text{if } \nu = \varepsilon \text{ then } \Gamma \vdash \tau_2 \in \mathsf{Seqable}) \quad \Gamma \vdash t :: \forall^\nu \alpha.\tau_1 \quad \tau_1[\tau_2/\alpha] \preceq \tau_3}{\Gamma \vdash (t_{\tau_2}) :: \tau_3} \; (\textsc{TApp''}_\nu)_{\nu \in \{\circ, \varepsilon\}}$$

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

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

Figure 12: The Typing Rules in **PolySeq**$^+$.

PolySeq$^+$. Its typing rules are given in Figure 12. For brevity we use the already introduced parameterization of rules.

Comparing the typing rules of **PolySeq*** and **PolySeq$^+$**, we can show that the terms typable in **PolySeq*** are also typable in **PolySeq$^+$** and vice versa.

Lemma 4.1. *If t is typable to τ under Γ in **PolySeq***, then it is typable to the same type τ under the same environment Γ in **PolySeq$^+$**, and conversely.*

For the proof of Lemma 4.1, we need the following auxiliary lemma.

Lemma 4.2. *The subtype relation $\preceq$, given through the rules in Figure 11, is reflexive and transitive.*

Proof. First we prove reflexivity. We use induction on the type structure of τ. If τ is a type variable, it follows immediately by the axiom (S-Var). For $\forall^\nu \alpha.\tau$ ($\nu = \circ, \varepsilon$), it is by (S-All$_{\nu,\nu}$) and the induction hypothesis. Similarly, we have reflexivity for types of the structure $\tau_1 \to^\nu \tau_2$ ($\nu = \circ, \varepsilon$), using (S-Arrow$_{\nu,\nu}$), and for $[\tau]$, using (S-List).

For transitivity we do an induction on the subtyping rules. For (S-Var) the result is immediate. Regarding (S-Arrow$_{\nu,\nu'}$) we have $\tau_1 \to^{\nu_a} \tau_1' \preceq \tau_2 \to^{\nu_b} \tau_2'$ and $\tau_2 \to^{\nu_b} \tau_2' \preceq \tau_3 \to^{\nu_c} \tau_3'$ as premises. By applying (S-Arrow$_{\nu,\nu'}$) backwards, we get: $\tau_2 \preceq \tau_1, \tau_1' \preceq \tau_2', \nu_b \leqslant \nu_a$ and $\tau_3 \preceq \tau_2, \tau_2' \preceq \tau_3', \nu_c \leqslant \nu_b$. Regarding the induction hypotheses and the already mentioned order $\circ < \varepsilon$ on $\{\circ, \varepsilon\}$, we have $\tau_3 \preceq \tau_1, \tau_1' \preceq \tau_3', \nu_c \leqslant \nu_a$ and apply (S-Arrow$_{\nu,\nu'}$) forwards to finish the proof.

The same scheme works for (S-All$_{\nu,\nu'}$) and (S-List).

Proof (of Lemma 4.1). First assume t is typable to τ under Γ in **PolySeq***. Then there is a derivation tree $\mathcal{T}$ leading to $\Gamma \vdash t :: \tau$. We give, inductively over the number of typing rules in $\mathcal{T}$ different from (Sub), a translation of $\mathcal{T}$ into a valid derivation tree $\mathcal{T}^+$ in **PolySeq$^+$**. For this it suffices to regard only the root of the derivation tree until the first appearance of a rule different from (Sub). Note that by transitivity of $\preceq$, stated in Lemma 4.2, we can transform a series of (Sub) rules into a single (Sub) rule and by reflexivity of $\preceq$, also stated in Lemma 4.2, we can assume the root of each derivation tree $\mathcal{T}$ to be (Sub). Hence, it remains only to consider the case with (Sub) the root of the derivation tree $\mathcal{T}$ preceded by a rule T different from (Sub). We consider all different possibilities for T, giving a rule transformation for each sequence of T and (Sub).

Regarding (Var) and (Nil) we add the restrictions as shown in Figure 12. This clearly replaces the application of (Sub) after these rules.

In the case (Cons) we 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

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

The replacement of τ'' by $[\tau']$ is not a limitation since, by the subtyping rules, only types with the same structure can be in a subtype relationship (cf. Lemma 3.4).

In the case (LCase) we replace

$$\dfrac{\dfrac{\Gamma \vdash t :: [\tau_1] \qquad \Gamma \vdash t_1 :: \tau_2 \qquad \Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau_2}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}) :: \tau_2}\,(\textsc{LCase}) \qquad \tau_2 \preceq \tau_2'}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}) :: \tau_2'}\,(\textsc{Sub})$$

by

$$\dfrac{\Gamma \vdash t :: [\tau_1] \qquad \dfrac{\Gamma \vdash t_1 :: \tau_2 \qquad \tau_2 \preceq \tau_2'}{\Gamma \vdash t_1 :: \tau_2'}\,(\textsc{Sub}) \qquad \dfrac{\Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau_2 \qquad \tau_2 \preceq \tau_2'}{\Gamma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau_2'}\,(\textsc{Sub})}{\Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{[\,] \to t_1\ ;\ x_1 : x_2 \to t_2\}) :: \tau_2'}\,(\text{L}\textsc{Case})$$

For the rules ($\textsc{Abs}_\nu$), $\nu \in \{\circ, \varepsilon\}$, we transform

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

into

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

Regarding the rule (S-$\textsc{Arrow}_{\nu,\nu'}$), we see that the two versions are equivalent. In the first version the conclusion of (S-$\textsc{Arrow}_{\nu,\nu'}$) is used and in the second one the premises are checked.

In the cases ($\textsc{App}_\nu$), $\nu \in \{\circ, \varepsilon\}$, we transform

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}\,(\textsc{App}_\nu) \qquad \tau_2 \preceq \tau_2'}{\Gamma \vdash (t_1\ t_2) :: \tau_2'}\,(\textsc{Sub})$$

into

$$\dfrac{\dfrac{\Gamma \vdash t_1 :: \tau_1 \to^\nu \tau_2 \qquad \tau_1 \to^\nu \tau_2 \preceq \tau_1 \to^\nu \tau_2'}{\Gamma \vdash t_1 :: \tau_1 \to^\nu \tau_2'}\,(\textsc{Sub}) \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2'}\,(\textsc{App}_\nu)$$

By the subtyping rule (S-$\textsc{Arrow}_{\nu,\nu}$) and reflexivity of subtyping, the different subtyping conditions in the original and the transformed derivation tree parts are equivalent.

For ($\text{T}\textsc{Abs}_\nu$), $\nu \in \{\circ, \varepsilon\}$, we transform

$$\dfrac{\dfrac{\alpha^\nu, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda\alpha.t) :: \forall^\nu \alpha.\tau}\,(\text{T}\textsc{Abs}_\nu) \qquad \forall^\nu \alpha.\tau \preceq \forall^{\nu'}.\tau'}{\Gamma \vdash (\Lambda\alpha.t) :: \forall^{\nu'}.\tau'}\,(\textsc{Sub})$$

into

$$\dfrac{\dfrac{\alpha^{\nu'}, \Gamma \vdash t :: \tau \qquad \tau \preceq \tau'}{\alpha^{\nu'}, \Gamma \vdash t :: \tau'}\,(\textsc{Sub})}{\Gamma \vdash (\Lambda\alpha.t) :: \forall^{\nu'}\alpha.\tau'}\,(\text{T}\textsc{Abs}_{\nu'})$$

where we set the subtype condition in the original tree to $\forall^\nu \alpha.\tau \preceq \forall^{\nu'}\alpha.\tau'$, which is no restriction for $\forall^\nu\alpha.\tau$ is fix and (S-$\textsc{All}_{\nu,\nu'}$), with $\nu \leqslant \nu'$, is the only suitable subtyping rule, able to fire only if the supertype is of the given structure. The fact $\nu \leqslant \nu'$ is used to validate the transformation, for it excludes the configuration with $\nu = \varepsilon$ and $\nu' = \circ$. At first sight it seems that we lost the former option of α° in the premise and $\forall^\varepsilon\alpha$ in the conclusion. But as all terms typable under α°, Γ are also typable under $\alpha^\varepsilon, \Gamma$ (α is only needed in the $\mathsf{Seqable}$-check), there is nothing lost.

In the cases ($\text{T}\textsc{App'}_\nu$), $\nu \in \{\circ, \varepsilon\}$, we have to introduce a new premise and transform

$$\dfrac{\dfrac{(\text{if } \nu = \varepsilon \text{ then } \Gamma \vdash \tau_2 \in \mathsf{Seqable}) \qquad \Gamma \vdash t :: \forall^\nu \alpha.\tau_1}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]}\,(\text{T}\textsc{App'}_\nu) \qquad \tau_1[\tau_2/\alpha] \preceq \tau'}{\Gamma \vdash (t_{\tau_2}) :: \tau'}\,(\textsc{Sub})$$

into

$$\dfrac{(\text{if } \nu = \varepsilon \text{ then } \Gamma \vdash \tau_2 \in \mathsf{Seqable}) \qquad \Gamma \vdash t :: \forall^\nu \alpha.\tau_1 \qquad \tau_1[\tau_2/\alpha] \preceq \tau'}{\Gamma \vdash (t_{\tau_2}) :: \tau'}\,(\text{T}\textsc{App''}_\nu)$$

Regarding ($\textsc{Fix}_\nu$), $\nu \in \{\circ, \varepsilon\}$, we melt ($\textsc{Sub}$) together with ($\textsc{Fix}_\nu$) by transforming

$$\dfrac{\dfrac{\Gamma \vdash t :: \tau \to^{\nu} \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau}\,(\textsc{Fix}_\nu) \qquad \tau \preceq \tau'}{\Gamma \vdash \mathbf{fix}\ t :: \tau'}\,(\textsc{Sub})$$

into

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

Finally, in the case ($\textsc{SLet'}$) we transform

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

into

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

To transform a valid derivation tree in **PolySeq**$^+$ into a valid one in **PolySeq***, we leave the rules that are equal in both rule sets unchanged and use the just given transformations backwards to replace the changed rules back by the original ones in **PolySeq*** combined with an application of ($\textsc{Sub}$).

This completes the proof.

With the calculus **PolySeq**$^+$ we made a big step towards a calculus with a typing rule system suitable to set up a type refinement algorithm. But there still remain some difficulties due to the fact that one term is usually typable to more than one refined type. That causes a whole family of derivation trees for a single term. Hence, **PolySeq**$^+$ is only an intermediate step on the way to a calculus whose typing rule system is directly interpretable as a type refinement algorithm. A calculus whose typing rule statements assign all refined types to a term at one time is presented in the next section.

5 PolySeq^C- Getting All Permissible Types

In the previous sections we presented the calculi **PolySeq*** and **PolySeq**$^+$, both allowing refined typing for all terms typable in the original calculus **PolySeq**. The typing rules of these calculi are declarative, but the intended use will be an algorithm that takes a closed term t with standard type annotations as in **PolySeq** and returns all, or better all minimal (in the sense of the minimal logical relation, and hence the strongest free theorems) refined types t is typable to (to be more accurate: for some choice of marks in the type annotations of t). Or, more generally, the same setting with t closed under a given standard **PolySeq** typing environment Γ has to be handled.

One way to solve this problem is to add concrete marks to Γ and t. Then we can regard **PolySeq**$^+$ as an algorithm that takes a typing environment Γ and a term t with concrete marks added and that tries to find all possible typing derivations for t under Γ by applying the typing rules of **PolySeq**$^+$ backwards. Afterwards, it constructs the sought-after types by the just constructed typing derivations. There will be multiple typing derivations since sometimes we have the choice between two rules, one introducing ε, the other $\circ$, as a mark. Also the construction of different sub- and supertypes by the subtyping system leads to multiple derivations. Additionally, we would have to run the algorithm with all possible initial choices for marks on the given standard Γ and t. To make a long story short: doing things this way seems to be very cumbersome.

Alternatively, avoiding the production of many trees and several runs with different inputs, we can switch to variable marks at type variables in Γ and at all arrows and $\forall$-quantifiers in each type annotation in Γ and t, and in the constructed type τ. This particularly avoids the parameterization

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

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

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

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

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

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

$$\frac{\begin{array}{c}\langle \dot\Gamma \vdash \dot\tau_2 \in \mathsf{Seqable} \rangle \Rrightarrow C_1 \\ \langle \dot\Gamma \vdash \dot t \rangle \Rrightarrow (C_2, \forall^\nu \alpha.\dot\tau_1) \qquad \langle \dot\tau_1[\dot\tau_2/\alpha] \preceq \cdot \rangle \Rrightarrow (C_3, \dot\tau_3)\end{array}}{\langle \dot\Gamma \vdash (\dot t_{\dot\tau_2}) \rangle \Rrightarrow (((\nu = \varepsilon) \Rightarrow (C_1)) \wedge C_2 \wedge C_3, \dot\tau_3)} \ (\textsc{TApp}^C)$$

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

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

Figure 13: The Conditional Typing Rules in **PolySeq**^{C}.

of rules as used in **PolySeq*** and **PolySeq**$^+$ and thereby eliminates all competition between different rules, allowing the interpretation of the arising rule system as a *deterministic* algorithm.

This solution is realized by the calculus **PolySeq**^{C}, presented in the current section, which has the same set of typable terms as **PolySeq**$^+$ and **PolySeq***, but in the first place 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 items are dotted to be distinguishable from concrete ones.

A statement about conditional typability is 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 $\nu \in V$, where V is a countable set of variables, disjoint from the sets of term and type variables. The typing rules for conditional typability are given in Figure 13 and rules of the subsystems, stating conditional class membership in $\mathsf{Seqable}$, subtyping, and equality, are shown in Figures 14, 15, and 16.

Although the rules are still declarative, they are now written in a slightly different style with all statements structured as *input* $\Rrightarrow$ *output* to emphasize their algorithmic nature and the intended use. Note that due to this style, the former subtyping rules are now split into rules for subtype

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

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

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

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

Figure 14: The Conditional Class Membership Rules for Seqable in **PolySeq**^{C}.

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

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

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

$$\frac{\langle \dot{\tau_1} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau_2})}{\langle \forall^{\nu} \alpha.\dot{\tau_1} \preceq \cdot \rangle \Rrightarrow (C \wedge (\nu \leqslant \nu'), \forall^{\nu'} \alpha.\dot{\tau_2})} \ (\text{S-All}_1^C)$$

$$\frac{\langle \cdot \preceq \dot{\tau_2} \rangle \Rrightarrow (C, \dot{\tau_1})}{\langle \cdot \preceq \forall^{\nu'} \alpha.\dot{\tau_2} \rangle \Rrightarrow (C \wedge (\nu \leqslant \nu'), \forall^{\nu} \alpha.\dot{\tau_1})} \ (\text{S-All}_2^C)$$

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

Figure 15: The Conditional Subtyping Rules in **PolySeq**^{C}.

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

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

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

Figure 16: The Conditional Equality Rules in **PolySeq**^{C}.

and for supertype search, both with constraint generation. Also equality of types, not explicitly present as a rule system before, is now expressed as conditional equality (cf. Figure 16).

Since terms, types, and typing environments in $\mathbf{PolySeq}^C$ are parameterized at the marks, we need a way of instantiating them to gain concrete items. The next definition introduces the *mark replacement* mapping for instantiation and also makes the notions *concrete* and *parameterized* precise.

Definition 5.1. A term, type, or typing environment is called *parameterized* if all marks at type variables, $\forall$-quantifiers, and arrows are variables. It is called *concrete* if all the marks are concrete, i.e. either $\circ$ or ε.

A mapping $\varrho : V_f \to L$ from a finite subset V_f of the set of variables V into the set of marks $L = \{\circ, \varepsilon\}$ is called a *mark replacement.*

With the help of mark replacements, we define typability in $\mathbf{PolySeq}^C$. Note that we keep the convention that concrete items are denoted without a dot on top, while parameterized items are dotted. Hence, both can be distinguished from each other and we do not explicitly mention whether they are concrete or parameterized.

Definition 5.2 (typability in $\mathbf{PolySeq}^C$). A term t is *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 $\mathbf{PolySeq}^C$.

Knowing how to define typability in $\mathbf{PolySeq}^C$, we can state the main requirement for the new calculus: typability in $\mathbf{PolySeq}^C$ agrees with typability in $\mathbf{PolySeq}^+$.

Theorem 5.3. *A term t is typable to a type τ under a typing environment Γ in **PolySeq**^{C} iff it is typable to the same τ under the same Γ in **PolySeq**$^+$.*

Apart from an example of the algorithmic use of the typing rules of $\mathbf{PolySeq}^C$ towards the end, the remaining part of this section deals with the proof of Theorem 5.3.

The proof is split into the two directions. For each, we break it down into a couple of lemmas, each regarding a subsystem of the typing rules. But beforehand, some more properties of mark replacements and parameterized items need to be defined, and some auxiliary lemmas are required.

Definition 5.4. Let $\dot{\kappa}$ be a parameterized term, type, or typing environment. The set of all variables used as marks in $\dot{\kappa}$ or a constraint C is called *mark variable set* of $\dot{\kappa}$ or C, and is denoted by $v(\dot{\kappa})$ or $v(C)$, respectively. Furthermore, $\dot{\kappa}$ is said to be *general* if each variable in $v(\dot{\kappa})$ occurs just once in $\dot{\kappa}$.

If $v(\dot{\kappa}) \subseteq \text{dom}(\varrho)$ for a mark replacement ϱ, we say that $\dot{\kappa}$ is *closed under* ϱ. A mark replacement ϱ is called *minimal* with respect to $\dot{\kappa}$ if $v(\dot{\kappa}) = \text{dom}(\varrho)$.

Similarly, we call a constraint C closed under ϱ if $v(C) \subseteq \text{dom}(\varrho)$, and ϱ minimal with respect to C if $v(C) = \text{dom}(\varrho)$.

If $\dot{\kappa}$ (C) is closed under a mark replacement ϱ, the result of the application of ϱ to $\dot{\kappa}$ (C) is called an *instantiation* of $\dot{\kappa}$ (C) by ϱ, denoted by $\dot{\kappa}_\varrho$ (C_ϱ), leading to a concrete term, type, or typing environment (propositional logic sentence), respectively.

If κ is a concrete term, type, or typing environment, then we call $\dot{\kappa}$ a *parameterization of* κ if there exists a mark replacement ϱ with $\dot{\kappa}_\varrho = \kappa$.

Two parameterized terms, types, or typing environments $\dot{\kappa}$ and $\dot{\kappa'}$ are *disjoint* if $v(\dot{\kappa}) \cap v(\dot{\kappa'}) = \emptyset$. Similarly, two mark replacements ϱ and ϱ' are *disjoint* if their domains are. They are *compatible* if there exists no $\nu \in V$ such that $\nu \in \text{dom}(\varrho) \cap \text{dom}(\varrho')$ and $\varrho(\nu) \neq \varrho'(\nu)$.

Let ϱ_1 and ϱ_2 be two compatible mark replacements. The mark replacement $\varrho = \varrho_1 \cup \varrho_2$ is defined as the union of the graphs of ϱ_1 and ϱ_2 and is called *union* of ϱ_1 and ϱ_2.

Lemma 5.5. *For all concrete terms, types, or typing environments $\kappa_1, \ldots, \kappa_n$, $n \in \mathbb{N}$, there exist general, disjoint parameterizations $\dot{\kappa}, \ldots, \dot{\kappa_n}$ and a mark replacement ϱ, such that $(\dot{\kappa_1})_\varrho = \kappa_1, \ldots, (\dot{\kappa_n})_\varrho = \kappa_n$.*

Proof. We construct $\dot{\kappa_1}, \ldots, \dot{\kappa_n}$ and the graph of ϱ by starting with $\varrho := \emptyset$ and replacing each concrete mark m in κ_1 by a variable ν with $\nu \notin \text{dom}(\varrho)$, extending ϱ by $\{\nu \mapsto m\}$. Afterwards we repeat this for $\kappa_2, \ldots, \kappa_n$, taking again only variables not already present in $\text{dom}(\varrho)$.

Now we can start considering the different subsystems necessary in a typing derivation. First, we analyze the rule system for conditional class membership in Seqable, shown in Figure 14.

Lemma 5.6. *For every parameterized* $\dot{\Gamma}$*, and* $\dot{\tau}$ *closed under* $\dot{\Gamma}$*, there exists some* C *with* $\langle \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$ *and* $v(C) \subseteq v(\dot{\Gamma}) \cup v(\dot{\tau})$*.*

Proof. The proof is by induction on the type structure of $\dot{\tau}$.

Lemma 5.7. *For all* Γ, τ *with* $\Gamma \vdash \tau \in \mathsf{Seqable}$*, there exist pairwise disjoint, general* $\dot{\Gamma}$ *and* $\dot{\tau}$*, a constraint* C*, and a mark replacement* ϱ*, such that* $\dot{\Gamma}_\varrho = \Gamma$*,* $\dot{\tau}_\varrho = \tau$*,* $C_\varrho = \mathit{True}$*, and* $\langle \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$*.*

Proof. We prove over the depth of the derivation tree of $\Gamma \vdash \tau \in \mathsf{Seqable}$, regarding only the last derivation rule, trying to translate it into a conditional class membership rule in **PolySeq**^{C}. For (C-List), Lemma 5.5 suffices to get a parameterization that fulfills (C-List^C) and also to obtain a suitable ϱ.

In the case (C-Arrow) we use again Lemma 5.5 and extend the gained ϱ by a new entry, mapping the variable ν (w.l.o.g. not yet in the domain) to ε.

For (C-Var) we take a general parameterization $\dot{\Gamma}$ of Γ and a corresponding mark replacement ϱ. By the premise of (C-Var), it has to map the mark variable at α to ε. Hence, we have (C-Var^C) in **PolySeq**^{C} as derivation for $\langle \dot{\Gamma} \vdash \alpha \in \mathsf{Seqable}\rangle \Rrightarrow (\nu = \varepsilon)$ and $(\nu = \varepsilon)_\varrho = \text{True}$.

Regarding (C-TAbs$_\varepsilon$), we consider the premise $\alpha^\varepsilon, \Gamma \vdash \tau \in \mathsf{Seqable}$. By the induction hypothesis, we know that there exist $\langle \alpha^\varepsilon, \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$ and ϱ^p with $\dot{\Gamma}_{\varrho^p} = \Gamma$, $\dot{\tau}_{\varrho^p} = \tau$, $C_{\varrho^p} = \text{True}$.

By (C-TAbs^C) we also have $\langle \dot{\Gamma} \vdash \forall \alpha^\nu.\dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$, and w.l.o.g. we can assume $\nu \notin \text{dom}(\varrho^p)$ and take $\varrho = \varrho^p \cup \{\nu \mapsto \varepsilon\}$. The case (CTAbs$_\circ$) is similar.

Lemma 5.8. *For all* $\dot{\Gamma}$*,* $\dot{\tau}$*,* C*, and* ϱ *with* $\langle \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$*, dom*$(\varrho) \supseteq v(\dot{\Gamma}) \cup v(\dot{\tau})$*, and* $C_\varrho = \mathit{True}$*, we have* $\dot{\Gamma}_\varrho \vdash \dot{\tau}_\varrho \in \mathsf{Seqable}$*.*

Proof. The proof is by induction on the derivation tree of $\langle \dot{\Gamma} \vdash \dot{\tau} \in \mathsf{Seqable}\rangle \Rrightarrow C$, regarding just the rule at the root node, translating it into a class membership rule from Figure 8 that is applicable for the concrete instantiations of $\dot{\Gamma}$ and $\dot{\tau}$ obtained by the application of each permitted ϱ, respectively. For (C-List^C) the result is immediate by taking (C-List). Regarding (C-Var^C), it can be translated into (C-Var), since $\varrho(\nu) = \varepsilon$, and hence the premise of (C-Var), is ensured by $C_\varrho = \text{True}$. The same holds for (C-Arrow^C) when translated to (C-Arrow).

For (C-TAbs^C) there are two possibilities, since $\nu \notin \text{dom}(C)$, and hence it can be mapped either to $\circ$ or to ε by a ϱ with $C_\varrho = \text{True}$. But at least, by $\text{dom}(\varrho) \supseteq v(\dot{\Gamma}) \cup v(\dot{\tau})$, it has to be in the domain of each ϱ under consideration. If ϱ maps ν to ε, we can translate (C-TAbs^C) into (C-TAbs), otherwise into (C-TAbs$_\circ$). A last important fact is that the premise of (C-TAbs^C) does not have a variable mark at α in the typing environment, and thus the typing environment does not satisfy our definition of "parameterized". We can solve this by replacing the ε-mark at α by ν', and substituting C by $C' := C \wedge (\nu' = \varepsilon)$.

Next, we regard the system for subtyping, shown in Figure 15.

Lemma 5.9. *For all* τ*,* τ' *with* $\tau \preceq \tau'$ *there exist general, disjoint* $\dot{\tau}$*,* $\dot{\tau'}$*, a constraint* C*, and a mark replacement* ϱ*, such that* $\dot{\tau}_\varrho = \tau$*,* $\dot{\tau'}_\varrho = \tau'$*,* $C_\varrho = \mathit{True}$*,* $\langle \dot{\tau} \preceq \cdot\rangle \Rrightarrow (C, \dot{\tau'})$*, and* $\langle \cdot \preceq \dot{\tau'}\rangle \Rrightarrow (C, \dot{\tau})$ *hold.*

Proof. The proof is by induction on the depth of the derivation tree of $\tau \preceq \tau'$ (cf. Figure 11). Hence, we regard just the rule at the root node, assuming Lemma 5.9 holds for all premises of that rule. The case (S-VAR) is immediate by Lemma 5.5 and (S-VAR$_1^C$) and (S-VAR$_2^C$), because $C_\varrho = \text{True}$ independently of the concrete mark replacement ϱ.

In all other cases the induction hypotheses suffice. As example, we regard the rule family $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu' \leqslant \nu}$. By the induction hypotheses, we have $\langle \cdot \preceq \dot{\sigma_1} \rangle \Rrightarrow (C_1, \dot{\tau_1})$, $\langle \dot{\tau_1} \preceq \cdot \rangle \Rrightarrow (C_1, \dot{\sigma_1})$, $\langle \dot{\sigma_2} \preceq \cdot \rangle \Rrightarrow (C_2, \dot{\tau_2})$, and $\langle \cdot \preceq \dot{\tau_2} \rangle \Rrightarrow (C_2, \dot{\sigma_2})$, which are exactly the premises for (S-ARROW$_1^C$) and (S-ARROW$_2^C$). Further we know ϱ and ϱ', w.l.o.g. disjoint, such that Lemma 5.9 is fulfilled for both premises. Hence, if we apply (S-ARROW$_1^C$) and (S-ARROW$_2^C$), build the union of ϱ and ϱ', and add the appropriate entries for ν and ν', we are done. The new constraint $C_1 \wedge C_2 \wedge (\nu' \leqslant \nu)$ is obviously fulfilled for all three concrete $(\text{S-ARROW}_{\nu,\nu'})_{\nu,\nu' \in \{\circ,\varepsilon\},\ \nu' \leqslant \nu}$ rules.

Lemma 5.10. *For all $\dot{\tau}$, $\dot{\tau'}$, C, and ϱ with $\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})$ or $\langle \cdot \preceq \dot{\tau'} \rangle \Rrightarrow (C, \dot{\tau})$, dom$(\varrho) \supseteq v(\dot{\tau}) \cup v(\dot{\tau'})$, and $C_\varrho = \text{True}$, we have $\dot{\tau}_\varrho \preceq \dot{\tau'}_\varrho$.*

Proof. Induction over the depth of the derivation tree for $\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})$ or $\langle \cdot \preceq \dot{\tau'} \rangle \Rrightarrow (C, \dot{\tau})$.

For the system of equality checks, shown in Figure 16, we state the next two lemmas.

Lemma 5.11. *For all τ, τ' with $\tau = \tau'$, there exist general, disjoint parameterizations $\dot{\tau}$, $\dot{\tau'}$, a constraint C, and a mark replacement ϱ, such that $\dot{\tau}_\varrho = \tau$, $\dot{\tau'}_\varrho = \tau'$, $C_\varrho = \text{True}$, and $\langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow C$.*

Proof. The proof is by induction on the type structure of τ, which is also the type structure of τ'. For τ being a type variable, the lemma is immediately true by (E-VAR^C) and $\varrho = \emptyset$.

Because all other cases are very similar to each other, we regard only $\tau = \tau_1 \to^\circ \tau_2$, and hence $\tau' = \tau_1' \to^\circ \tau_2'$. By the induction hypotheses, there exist $\dot{\tau_1}$, $\dot{\tau_1'}$, $\dot{\tau_2}$, $\dot{\tau_2'}$, and ϱ_1, ϱ_2, w.l.o.g. disjoint, such that $\langle \dot{\tau_1} = \dot{\tau_1'} \rangle \Rrightarrow C_1$ and $\langle \dot{\tau_2} = \dot{\tau_2'} \rangle \Rrightarrow C_2$ hold, as well as $(\dot{\tau_1})_{\varrho_1} = \tau_1$, $(\dot{\tau_1'})_{\varrho_1} = \tau_1'$, $(\dot{\tau_2})_{\varrho_2} = \tau_2$, $(\dot{\tau_2'})_{\varrho_2} = \tau_2'$, $(C_1)_{\varrho_1} = \text{True}$, and $(C_2)_{\varrho_2} = \text{True}$. Hence, we can apply (E-ARROW^C). Taking $\varrho = \varrho_1 \cup \varrho_2$ extended by the entries $\nu \mapsto \circ$ and $\nu' \mapsto \circ$, we are done.

Lemma 5.12. *For all $\dot{\tau}$, $\dot{\tau'}$, C, and ϱ with $\langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow C$, dom$(\varrho) \supseteq v(\dot{\tau}) \cup v(\dot{\tau'})$, and $C_\varrho = \text{True}$, we have $\dot{\tau}_\varrho = \dot{\tau'}_\varrho$.*

Proof. Induction over the depth of the derivation tree of $\langle \dot{\tau} = \dot{\tau'} \rangle \Rrightarrow C$.

Now we focus on the main typing rule system, shown in Figure 13.

Lemma 5.13. *For all concrete t, τ, Γ with $\Gamma \vdash t :: \tau$ in **PolySeq**$^+$, there exist pairwise disjoint, general parameterizations $\dot{\Gamma}$, $\dot{t}$, $\dot{\tau}$, a constraint C, and a mark replacement ϱ such that $\dot{\Gamma}_\varrho = \Gamma$, $\dot{t}_\varrho = t$, $\dot{\tau}_\varrho = \tau$, $C_\varrho = \text{True}$, and $\langle \dot{\Gamma} \vdash \dot{t} \rangle \Rrightarrow (C, \dot{\tau})$ in **PolySeq**^{C}.*

Proof. We prove by induction over the depth of the derivation tree of $\Gamma \vdash t :: \tau$ in **PolySeq**$^+$. Hence, we regard only the last derivation rule.

In the case

$$\frac{\tau \preceq \tau'}{\Gamma, x :: \tau \vdash x :: \tau'}\ (\text{VAR'})\,,$$

by Lemma 5.9 we have $\dot{\tau}$, $\dot{\tau'}$, C, and ϱ^p, such that $\langle \dot{\tau} \preceq \cdot \rangle \Rrightarrow (C, \dot{\tau'})$, $C_{\varrho^p} = \text{True}$, $\dot{\tau}_{\varrho^p} = \tau$, and $\dot{\tau'}_{\varrho^p} = \tau'$. We can find a general parameterization $\dot{\Gamma}$ of Γ, such that $v(\dot{\Gamma}) \cap \text{dom}(\varrho^p) = \emptyset$ and a minimal mark replacement ϱ' with respect to $\dot{\Gamma}$, such that $\dot{\Gamma}_{\varrho'} = \Gamma$. Now with $\varrho = \varrho^p \cup \varrho'$ it holds that $C_\varrho = \text{True}$, $\dot{\Gamma}_\varrho = \Gamma$, $\dot{\tau}_\varrho = \tau$, and $\dot{\tau'}_\varrho = \tau'$. By (VAR^C) we have $\langle \dot{\Gamma}, x :: \dot{\tau} \vdash x \rangle \Rrightarrow (C, \dot{\tau'})$ and are done.

The case (NIL') is similar to (VAR').

In the case

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

we rewrite the rule to

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: [\tau'] \qquad \tau = \tau'}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \text{ (CONS)}.$$

By the induction hypotheses and Lemma 5.11 we have general parameterizations $\dot\Gamma_1$ and $\dot\Gamma_2$ of Γ, $\dot\tau_1$ and $\dot\tau_2$ of τ, as well as $\dot{\tau'}_1$ and $\dot{\tau'}_2$ of τ'. W.l.o.g. we assume $\dot\Gamma = \dot\Gamma_1 = \dot\Gamma_2$, $\dot\tau = \dot\tau_1 = \dot\tau_2$, and $\dot{\tau'} = \dot{\tau'}_1 = \dot{\tau'}_2$. Furthermore, we have ϱ_1^p, ϱ_2^p, and ϱ_3^p such that

$$\langle \dot\Gamma \vdash \dot{t_1}\rangle \Rrightarrow (C_1, \dot\tau) \text{ holds and } (C_1)_{\varrho_1^p} = \text{True},\ (\dot\Gamma)_{\varrho_1^p} = \Gamma,\ (\dot{t_1})_{\varrho_1^p} = t_1,\ (\dot\tau)_{\varrho_1^p} = \tau\,;$$

$$\langle \dot\Gamma \vdash \dot{t_2}\rangle \Rrightarrow (C_2, [\dot{\tau'}]) \text{ holds and } (C_2)_{\varrho_2^p} = \text{True},\ (\dot\Gamma)_{\varrho_2^p} = \Gamma,\ (\dot{t_2})_{\varrho_2^p} = t_2,\ (\dot{\tau'})_{\varrho_2^p} = \tau'\,;$$

as well as

$$\langle \dot\tau = \dot{\tau'}\rangle \Rrightarrow C_3 \text{ holds and } (C_3)_{\varrho_3^p} = \text{True},\ (\dot\tau)_{\varrho_3^p} = \tau,\ (\dot{\tau'})_{\varrho_3^p} = \tau'\,.$$

We can assume $\mathrm{dom}(\varrho_1^p) \cap \mathrm{dom}(\varrho_2^p) = v(\dot\Gamma)$, $\mathrm{dom}(\varrho_1^p) \cap \mathrm{dom}(\varrho_3^p) = v(\dot\tau)$, and $\mathrm{dom}(\varrho_2^p) \cap \mathrm{dom}(\varrho_3^p) = v(\dot{\tau'})$. Therefore, all three are pairwise compatible and we take $\varrho = \varrho_1^p \cup \varrho_2^p \cup \varrho_3^p$. By applying (CONS^C) with the three just stated premises, we are done.

For the remaining rules, we will just point out differences from the already proved cases.

Regarding (ABS'$_\varepsilon$), we extend ϱ by a new entry $\nu \mapsto \varepsilon$. Similarly, for (ABS'$_\circ$) we add $\nu \mapsto \circ$ as new entry.

In the case (TAPP"$_\varepsilon$) Lemma 5.7 is used.

For

$$\frac{\Gamma \vdash t :: \forall^\circ\alpha.\tau_1 \qquad \tau_1[\tau_2/\alpha] \preceq \tau_3}{\Gamma \vdash (t_{\tau_2}) :: \tau_3} \text{ (TAPP"}_\circ)$$

we need that for each τ_2 and Γ with τ_2 closed under Γ and all general parameterizations $\dot\Gamma$ and $\dot\tau_2$ of Γ and τ_2, there exists some C_1 with $\langle \dot\Gamma \vdash \dot\tau \in \mathsf{Seqable}\rangle \Rrightarrow C_1$ and $v(C_1) \subseteq v(\dot\Gamma) \cup v(\dot\tau_2)$. This is true by Lemma 5.6. Hence, considering additionally the induction hypotheses, we can fire rule (TAPP^C) in **PolySeq**^{C} and take ϱ to be the union of the assignments known for the second and third premise, extended by $\nu \mapsto \circ$. For $v(C_1) \subseteq v(\dot\Gamma) \cup v(\dot\tau_2)$, C_1 is closed under ϱ, and for $(\circ = \varepsilon) \Leftrightarrow \text{False}$, the constraint $((\nu = \varepsilon) \Rightarrow (C_1)) \wedge C_2 \wedge C_3$ evaluates to True when instantiated by ϱ.

Lemma 5.14. *For all $\dot\Gamma$, $\dot t$, $\dot\tau$, constraints C, and assignments ϱ with $v(\dot\Gamma) \cup v(\dot t) \cup v(\dot\tau) \subseteq dom(\varrho)$, such that $\langle \dot\Gamma \vdash \dot t\rangle \Rrightarrow (C, \dot\tau)$ and $C_\varrho = True$, we have $\dot\Gamma_\varrho \vdash \dot t_\varrho :: \dot\tau_\varrho$ in **PolySeq**$^+$.*

Proof. The proof is by induction over the depth of the derivation tree of $\langle \dot\Gamma \vdash \dot t\rangle \Rrightarrow (C, \dot\tau)$ in **PolySeq**^{C}. We only consider the rule at the root node of the derivation tree and know that for each premise of that rule Lemma 5.14 holds either by the induction hypothesis, or the premise satisfies one of the Lemmas 5.8, 5.10, 5.12.

In the case (VAR^C) we have by Lemma 5.10 that $\dot\tau_\varrho \preceq \dot{\tau'}_\varrho$ holds for all ϱ with $v(\dot\tau) \cup v(\dot{\tau'}) \subseteq \mathrm{dom}(\varrho)$ and $C_\varrho = \text{True}$. Hence, this holds also for all ϱ with $v(\dot\tau) \cup v(\dot{\tau'}) \cup v(\dot\Gamma) \subseteq \mathrm{dom}(\varrho)$ and $C_\varrho = \text{True}$, and by (VAR') we are done.

The case (NIL^C) is similar to (VAR^C).

Regarding

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

we have

$$\forall \varrho \in \Psi_1.(\langle \dot\Gamma \vdash \dot{t_1}\rangle \Rrightarrow (C_1, \dot\tau)) \Rightarrow \dot\Gamma_\varrho \vdash (\dot{t_1})_\varrho :: \dot\tau_\varrho\,,$$

$$\forall \varrho \in \Psi_2.(\langle \dot\Gamma \vdash \dot{t_2}\rangle \Rrightarrow (C_2, [\dot{\tau'}])) \Rightarrow \dot\Gamma_\varrho \vdash (\dot{t_2})_\varrho :: [\dot{\tau'}]_\varrho\,,$$

and

$$\forall \varrho \in \Psi_3.(\langle \dot\tau = \dot{\tau'}\rangle \Rrightarrow C_3) \Rightarrow \dot\tau_\varrho = \dot{\tau'}_\varrho$$

with

$$\Psi_1 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot{t_1}) \cup v(\dot\tau) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1)_\varrho = \mathrm{True})\}$$
$$\Psi_2 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot{t_2}) \cup v(\dot{\tau'}) \subseteq \mathrm{dom}(\varrho) \wedge ((C_2)_\varrho = \mathrm{True})\}$$
$$\Psi_3 = \{\varrho \mid v(\dot\tau) \cup v(\dot{\tau'}) \subseteq \mathrm{dom}(\varrho) \wedge ((C_3)_\varrho = \mathrm{True})\}$$

by the induction hypothesis and Lemma 5.12.

Therefore, with a slight rewrite to an extra equality premise, we have all premises to apply (Cons) for all instantiations of $\dot\Gamma$, $\dot{t_1}$, $\dot{t_2}$, $\dot\tau$, $\dot{\tau'}$ by any $\varrho \in \Psi_1 \cap \Psi_2 \cap \Psi_3$. Now, we know that for all $\varrho \in \Psi_1 \cap \Psi_2 \cap \Psi_3 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot{t_1}) \cup v(\dot{t_2}) \cup v(\dot\tau) \cup v(\dot{\tau'}) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1 \wedge C_2 \wedge C_3)_\varrho = \mathrm{True})\}$, we have $\dot\Gamma_\varrho \vdash ((\dot{t_1})_\varrho : (\dot{t_2})_\varrho) :: [\tau]_\varrho$. Since we do not lose anything when restricting the domain of ϱ to the entries really used during the instantiation, we have the same result for all $\varrho \in \{\varrho \mid v(\dot\Gamma) \cup v(\dot{t_1}) \cup v(\dot{t_2}) \cup v(\dot\tau) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1 \wedge C_2 \wedge C_3)_\varrho = \mathrm{True})\}$, which is exactly the claim.

The proof for (LCase^C) is similar.

In the case

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

we have

$$\forall\varrho \in \Psi_1.(\langle\dot\Gamma, x :: \dot\tau_1 \vdash \dot t\rangle \Rrightarrow (C_1, \dot\tau_2) \Rightarrow \dot\Gamma_\varrho, x :: (\dot\tau_1)_\varrho \vdash \dot t_\varrho :: (\dot\tau_2)_\varrho$$

and

$$\forall\varrho \in \Psi_2.(\langle \cdot \preceq \dot\tau_1\rangle \Rrightarrow (C_2, \dot{\tau_1'})) \Rightarrow (\dot{\tau_1'})_\varrho \preceq (\dot\tau_1)_\varrho$$

with

$$\Psi_1 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot\tau_1) \cup v(\dot\tau_2) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1)_\varrho = \mathrm{True})\}$$

and

$$\Psi_2 = \{\varrho \mid v(\dot\tau_1) \cup v(\dot{\tau_1'}) \subseteq \mathrm{dom}(\varrho) \wedge ((C_2)_\varrho = \mathrm{True})\}$$

by the induction hypothesis and Lemma 5.10. Hence, we can apply either (Abs'$_\varepsilon$) or (Abs'$_\circ$) in **PolySeq$^+$**. For the first rule we get $\dot\Gamma_\varrho \vdash (\lambda x :: (\dot\tau_1)_\varrho.\dot t_\varrho) :: (\dot{\tau_1'} \to^\nu \dot\tau_2)_\varrho$ for all $\varrho \in \Psi_1 \cap \Psi_2 \cap \{\varrho \mid \varrho(\nu) = \varepsilon\}$. And for the second rule we get the same statement but for $\varrho \in \Psi_1 \cap \Psi_2 \cap \{\varrho \mid \varrho(\nu) = \circ\}$. Therefore, we have $\dot\Gamma_\varrho \vdash (\lambda x :: (\dot\tau_1)_\varrho.\dot t_\varrho) :: (\dot{\tau_1'} \to^\nu \dot\tau_2)_\varrho$ for all $\varrho \in \{\varrho \mid v(\dot\Gamma) \cup v(\dot\tau_1) \cup v(\dot t) \cup v(\dot{\tau_1'} \to^\nu \dot\tau_2) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1 \wedge C_2)_\varrho = \mathrm{True})\}$, which is the claim.

A remaining interesting case is

$$\frac{\begin{array}{c}\langle\dot\Gamma \vdash \dot\tau_2 \in \mathsf{Seqable}\rangle \Rrightarrow C_1 \\ \langle\dot\Gamma \vdash \dot t\rangle \Rrightarrow (C_2, \forall^\nu\alpha.\dot\tau_1) \qquad \langle\dot\tau_1[\dot\tau_2/\alpha] \preceq \cdot\rangle \Rrightarrow (C_3, \dot\tau_3)\end{array}}{\langle\dot\Gamma \vdash (\dot t_{\dot\tau_2})\rangle \Rrightarrow (((\nu = \varepsilon) \Rightarrow (C_1)) \wedge C_2 \wedge C_3, \dot\tau_3)} \ (\mathrm{TApp}^C)$$

where by Lemma 5.8, the induction hypothesis, and Lemma 5.10 we have

$$\forall\varrho \in \Psi_1.(\langle\dot\Gamma \vdash \dot\tau_2 \in \mathsf{Seqable}\rangle \Rrightarrow C_1) \Rightarrow \dot\Gamma_\varrho \vdash (\dot\tau_2)_\varrho \in \mathsf{Seqable}\,,$$

$$\forall\varrho \in \Psi_2.(\langle\dot\Gamma \vdash \dot t\rangle \Rrightarrow (C_2, \forall^\nu\alpha.\dot\tau_1)) \Rightarrow \dot\Gamma_\varrho \vdash \dot t_\varrho :: (\forall^\nu\alpha.\dot\tau_1)_\varrho\,,$$

and

$$\forall\varrho \in \Psi_3.(\langle\dot\tau_1[\dot\tau_2/\alpha] \preceq \cdot\rangle \Rrightarrow (C_3, \dot\tau_3)) \Rightarrow (\dot\tau_1)_\varrho[(\dot\tau_2)_\varrho/\alpha] \preceq (\dot\tau_3)_\varrho$$

with

$$\Psi_1 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot\tau_2) \subseteq \mathrm{dom}(\varrho) \wedge ((C_1)_\varrho = \mathrm{True})\}$$
$$\Psi_2 = \{\varrho \mid v(\dot\Gamma) \cup v(\dot t) \cup v(\dot\tau_1) \cup \{\nu\} \subseteq \mathrm{dom}(\varrho) \wedge ((C_2)_\varrho = \mathrm{True})\}$$
$$\Psi_3 = \{\varrho \mid v(\dot\tau_1) \cup v(\dot\tau_2) \cup \mid v(\dot\tau_3) \subseteq \mathrm{dom}(\varrho) \wedge ((C_3)_\varrho = \mathrm{True})\}\,.$$

If we decompose Ψ_2 into $\Psi_2^\circ = \{\varrho \mid \varrho \in \Psi_2 \wedge \varrho(\nu) = \circ\}$ and $\Psi_2^\varepsilon = \{\varrho \mid \varrho \in \Psi_2 \wedge \varrho(\nu) = \varepsilon\}$, we have $\dot\Gamma_\varrho \vdash (\dot t_\varrho)_{(\dot\tau_2)_\varrho} :: (\dot\tau_3)_\varrho$ for all $\varrho \in \Psi^\circ = \Psi_2^\circ \cap \Psi_3$ by (TApp"$_\circ$), and the same for $\varrho \in \Psi^\varepsilon = \Psi_1 \cap \Psi_2^\varepsilon \cap \Psi_3$ by (TApp"$_\varepsilon$). Hence, we have the statement for all $\varrho \in \Psi^\circ \cup \Psi^\varepsilon = \{\varrho \mid v(\dot\Gamma) \cup v(\dot t) \cup v(\dot\tau_1) \cup v(\dot\tau_2) \cup v(\dot\tau_3) \cup \{\nu\} \subseteq \mathrm{dom}(\varrho) \wedge ((((\nu = \varepsilon) \Rightarrow (C_1)) \wedge C_2 \wedge C_3)_\varrho = \mathrm{True})\}$, and by the same reasoning as in the case (Cons^C) we get the claim.

The proof of the remaining rules requires nothing not already mentioned.

Proof (of Theorem 5.3). Equivalence, in the sense of typability, is direct by Definition 5.2 and Lemmas 5.13 and 5.14.

With $\mathbf{PolySeq}^C$ we have reached the initial goal of a calculus with a refined type system, providing for stronger free theorems than the standard calculus **PolySeq**, and a system of typing rules suitable to set up an algorithm which returns the set of all refined types a term, given with standard type annotations as in **PolySeq**, is typable to.

By Definition 5.2 and Theorem 5.3, it suffices to investigate conditional typability as stated by the typing rule system from Figure 13 to capture all types a given term is typable to. Afterwards, we can solve the constraint and instantiate the parameterized typing environment, term, and type to gain the set of all refined types. Furthermore, by the subtype relation stated through the rules in Figure 11 and characterized by Lemma 3.7, we can cut down the set of refined types to the minimal ones, providing the strongest free theorems.

As an example how $\mathbf{PolySeq}^C$ is 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 typing environment is empty. First, we add pairwise distinct variable marks, $\nu_1, \ldots, \nu_m$, at all $\forall$-quantifiers and arrows at the type annotations in *foldl''*. This manipulation is denoted by a dot over $\dot{foldl''}$. Now we use the typing rules of $\mathbf{PolySeq}^C$ backwards to generate a derivation tree for $\dot{foldl''}$ in the empty 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 $\langle \vdash \dot{foldl''} \rangle \Rrightarrow (C, \dot{\tau})$ holds in $\mathbf{PolySeq}^C$. The parameterized type $\dot{\tau}$ contains variable marks $\nu_{m+1}, \ldots, \nu_{m+n}$, and C imposes constraints on the marks $\nu_1 \ldots \nu_{m+n}$ (and on other marks only used during the typing derivation). Now we determine the mark replacements ϱ that instantiate $\nu_{m+1}, \ldots, \nu_{m+n}$ and for which C_ϱ is True. Their application to $\dot{\tau}$ provides us all refined types of *foldl''*. In a last step, we sort out types that are not minimal in this set with respect to the subtype relation stated by the rules in Figure 11, because these types would induce unnecessary restrictions on the corresponding free theorems.

For *foldl''* we end up with the already known refined type $\forall^\circ a.\forall b.(a \to^\circ b \to a) \to a \to [b] \to a$. But now it has been derived automatically!

6 Implementation and Extensions

Up to now, the introduced polymorphic calculi contain only lists as algebraic data type, but the extension to other algebraic data types and base types like integers and Booleans is straightforward. Everything required is already present from the handling of lists. Also the introduction of operations like integer addition requires nothing new.

Extensions are done for integers (with addition) and Booleans (with a case-statement), implemented and usable through a web interface[7]. The extended term and type syntax is shown in Figure 17. When using the web interface, the ASCII syntax is `_{}` for subscripts, `->` for $\to$, `\` for λ, `/\` for Λ, and `forall` for $\forall$.

To allow more natural input, some syntactic sugar is added. We allow the statements

$$\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \quad \text{and} \quad \textbf{case } t \textbf{ of } \{\texttt{False} \to t_2\,;\texttt{True} \to t_1\}\,,$$

which are both translated into **case** t **of** $\{\texttt{True} \to t_1\,;\texttt{False} \to t_2\}$. Furthermore, we introduce a **seq** primitive[8] and a non-strict (and non-recursive) **let** statement.

The corresponding new type derivation rules are shown in Figure 18. Also the conditional subtyping, class membership, and equality rule systems, as well as the non-conditional subtyping rule system shown in Figure 11, are extended in a straightforward way.

The implementation takes a term with standard type annotations at all binding occurrences of term variables and returns all minimal (in the sense of minimal logical relation) refined types the

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

[8] Here **seq** is not a first-class function. Partial applications of it are not allowed.

$$\tau ::= \alpha \mid \alpha^\circ \mid [\tau] \mid \tau \to \tau \mid \tau \to^\circ \tau \mid \forall\alpha.\tau \mid \mathsf{Int} \mid \mathsf{Bool}$$

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

Figure 17: Syntax of Types τ and Terms t of the Extended Calculus.

$$\langle \Gamma \vdash n \rangle \Rrightarrow (\mathrm{True}, \mathsf{Int})\ (\textsc{Int}^C)$$

$$\langle \Gamma \vdash \texttt{True} \rangle \Rrightarrow (\mathrm{True}, \mathsf{Bool})\ (\textsc{True}^C) \qquad \langle \Gamma \vdash \texttt{False} \rangle \Rrightarrow (\mathrm{True}, \mathsf{Bool})\ (\textsc{False}^C)$$

$$\frac{\langle \Gamma \vdash t_1 \rangle \Rrightarrow (C_1, \mathsf{Int}) \qquad \langle \Gamma \vdash t_2 \rangle \Rrightarrow (C_2, \mathsf{Int})}{\langle \Gamma \vdash (t_1 + t_2) \rangle \Rrightarrow (C_1 \wedge C_2, \mathsf{Int})}\ (\textsc{Add}^C)$$

$$\frac{\begin{array}{c}\langle \Gamma \vdash t \rangle \Rrightarrow (C_1, \mathsf{Bool}) \qquad \langle \Gamma \vdash t_1 \rangle \Rrightarrow (C_2, \tau_2) \\ \langle \Gamma \vdash t_2 \rangle \Rrightarrow (C_3, \tau_2') \qquad \langle \tau_2 = \tau_2' \rangle \Rrightarrow C_4\end{array}}{\langle \Gamma \vdash (\textbf{case}\ t\ \textbf{of}\ \{\texttt{True} \to t_1\,;\texttt{False} \to t_2\}) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3 \wedge C_4, \tau_2)}\ (\textsc{BCase}^C)$$

$$\frac{\langle \Gamma \vdash t_1 \rangle \Rrightarrow (C_1, \tau_1) \qquad \langle \Gamma, x :: \tau_1 \vdash t_2 \rangle \Rrightarrow (C_2, \tau_2)}{\langle \Gamma \vdash (\textbf{let!}\ x = t_1\ \textbf{in}\ t_2) \rangle \Rrightarrow (C_1 \wedge C_2, \tau_2)}\ (\textsc{Let}^C)$$

$$\frac{\langle \Gamma \vdash t_1 \rangle \Rrightarrow (C_1, \tau_1) \qquad \langle \Gamma \vdash \tau_1 \in \mathsf{Seqable} \rangle \Rrightarrow C_2 \qquad \langle \Gamma \vdash t_2 \rangle \Rrightarrow (C_3, \tau_2)}{\langle \Gamma \vdash (\textbf{seq}\ t_1\ t_2) \rangle \Rrightarrow (C_1 \wedge C_2 \wedge C_3, \tau_2)}\ (\textsc{Seq}^C)$$

Figure 18: Additional Conditional Typing Rules for the Extension of $\mathbf{PolySeq}^C$.

term is typable to in the refined type system. Additionally, it presents the refined free theorems, as well as the theorem for the standard type. The interesting parts in the theorems are highlighted in the web interface.

Regarding the initial examples of *foldl* and the strict versions *foldl'*, *foldl''*, and *foldl'''*[9], the highlighted parts in the free theorems produced 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 restrictions vanish. (The remaining highlighted parts in this case represent a strengthening of the theorem, not a restriction.) A screenshot of the output for *foldl''* is shown in Figure 19. Altogether, it is possible to place and drop each of the strict let statements and analyze which statement forces which restriction. Thus, we really get a fine-grained analysis about how selective strictness influences free theorems.

7 Conclusion

The calculus developed, with the refined type system and the possibility to automatically retype standardly typed terms to refined types, allows a fine-grained analysis of the influence of selective strict evaluation on free theorems depending on its concrete use in a term. By the implementation of the retyping algorithm we provide a tool locating exactly which preconditions on a free theorem arise from particular uses of selective strict evaluation in a concrete term. Hence, we are able to regain the best reachable (equational) free theorems with respect to analyzing where strictness is forced.

[9] The functions' syntax must be adapted to the syntax used in the web interface, as just given. For an example, see the screenshot of the web interface in Figure 19.

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 19: Screenshot of the Web Interface's Output for *foldl''*.

The presented calculus **PolySeq**, especially considered with straightforward extension to algebraic data types and base types as described in Section 6, is quite close to real-world lazy functional programming languages like Haskell. This allows us to apply the refined free theorems for the correctness proofs of transformations in such languages. For example, the correctness of the fusion property for *foldl* mentioned in the introduction is 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 fusion property holds for *foldl′*.

References

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA, Proceedings*, pages 223–232. ACM Press, 1993. DOI: 10.1145/165180.165214.

P. Hudak, J. Hughes, S. L. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL-III, Proceedings*, pages 1–55, June 2007.

G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *POPL, Proceedings*, pages 99–110. ACM Press, 2004. DOI: 10.1145/982962.964010.

J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *ESOP, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996. DOI: 10.1007/3-540-61055-3_38.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

D. Seidel and J. Voigtländer. Checking the influence of non-termination on free theorems (Extended abstract). In *WST, Informal Proceedings*, pages 80–83, 2009.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP, Proceedings*, pages 124–132. ACM Press, 2002. DOI: 10.1145/583852.581491.

J. Voigtländer. Proving correctness via free theorems: The case of the destroy/build-rule. In *PEPM, Proceedings*, pages 13–20. ACM Press, 2008. DOI: 10.1145/1328408.1328412.

P. Wadler. Theorems for free! In *FPCA, Proceedings*, pages 347–359. ACM Press, 1989. DOI: 10.1145/99370.99404.

A Proof of Theorem 2.2

Proof. The proof is by induction over typing derivations with respect to the system from Figures 3 and 4. The cases $\Gamma, x :: \tau \vdash x :: \tau$ and $\Gamma \vdash [\,]_\tau :: [\tau]$ are immediate.

In the case

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

we have

$$\begin{array}{rl} & ([\![t_1 : t_2]\!]_{\theta_1,\sigma_1}, [\![t_1 : t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{[\tau],\rho} \\ \Leftrightarrow & (\lfloor [\![t_1]\!]_{\theta_1,\sigma_1} : [\![t_2]\!]_{\theta_1,\sigma_1} \rfloor, \lfloor [\![t_1]\!]_{\theta_2,\sigma_2} : [\![t_2]\!]_{\theta_2,\sigma_2} \rfloor) \in \mathit{list}\, \Delta_{\tau,\rho} \\ \Leftrightarrow & ([\![t_1]\!]_{\theta_1,\sigma_1}, [\![t_1]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho},\ ([\![t_2]\!]_{\theta_1,\sigma_1}, [\![t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{[\tau],\rho}\,, \end{array}$$

so the induction hypotheses suffice.

In the case

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

we have to show that the values

$$\begin{cases} [\![t_1]\!]_{\theta_1,\sigma_1} & \text{if } [\![t]\!]_{\theta_1,\sigma_1} = \lfloor [\,] \rfloor \\ [\![t_2]\!]_{\theta_1,\sigma_1[x_1 \mapsto a,\, x_2 \mapsto b]} & \text{if } [\![t]\!]_{\theta_1,\sigma_1} = \lfloor a : b \rfloor \\ \bot & \text{if } [\![t]\!]_{\theta_1,\sigma_1} = \bot \end{cases}$$

and

$$\begin{cases} [\![t_1]\!]_{\theta_2,\sigma_2} & \text{if } [\![t]\!]_{\theta_2,\sigma_2} = \lfloor [\,] \rfloor \\ [\![t_2]\!]_{\theta_2,\sigma_2[x_1 \mapsto c,\, x_2 \mapsto d]} & \text{if } [\![t]\!]_{\theta_2,\sigma_2} = \lfloor c : d \rfloor \\ \bot & \text{if } [\![t]\!]_{\theta_2,\sigma_2} = \bot \end{cases}$$

are related by $\Delta_{\tau_2,\rho}$. Since $([\![t]\!]_{\theta_1,\sigma_1}, [\![t]\!]_{\theta_2,\sigma_2}) \in \Delta_{[\tau_1],\rho} = \mathit{list}\ \Delta_{\tau_1,\rho}$ by induction hypothesis, we only have to consider the following three cases:

1. $[\![t]\!]_{\theta_1,\sigma_1} = \lfloor [\,] \rfloor$ and $[\![t]\!]_{\theta_2,\sigma_2} = \lfloor [\,] \rfloor$, in which case the induction hypothesis $([\![t_1]\!]_{\theta_1,\sigma_1}, [\![t_1]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_2,\rho}$ suffices,

2. $[\![t]\!]_{\theta_1,\sigma_1} = \lfloor a : b \rfloor$ and $[\![t]\!]_{\theta_2,\sigma_2} = \lfloor c : d \rfloor$ with $(a, c) \in \Delta_{\tau_1,\rho}$ and $(b, d) \in \mathit{list}\ \Delta_{\tau_1,\rho} = \Delta_{[\tau_1],\rho}$, in which case the induction hypothesis that for every such a, b, c, and d,
$$([\![t_2]\!]_{\theta_1,\sigma_1[x_1 \mapsto a,\, x_2 \mapsto b]}, [\![t_2]\!]_{\theta_2,\sigma_2[x_1 \mapsto c,\, x_2 \mapsto d]}) \in \Delta_{\tau_2,\rho}\,,$$
suffices, and

3. $[\![t]\!]_{\theta_1,\sigma_1} = \bot$ and $[\![t]\!]_{\theta_2,\sigma_2} = \bot$, in which case we have to show $(\bot, \bot) \in \Delta_{\tau_2,\rho}$, which follows from strictness of $\Delta_{\tau_2,\rho}$ (cf. Lemma 2.1).

In the case

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

we have

$$\begin{array}{ll} & ([\![\lambda x :: \tau_1.t]\!]_{\theta_1,\sigma_1}, [\![\lambda x :: \tau_1.t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1 \to \tau_2,\rho} \\ \Leftrightarrow & (\lfloor \lambda a.[\![t]\!]_{\theta_1,\sigma_1[x \mapsto a]} \rfloor, \lfloor \lambda b.[\![t]\!]_{\theta_2,\sigma_2[x \mapsto b]} \rfloor) \in \Delta_{\tau_1 \to \tau_2,\rho} \\ \Leftrightarrow & \forall (a, b) \in \Delta_{\tau_1,\rho}.\ ([\![t]\!]_{\theta_1,\sigma_1[x \mapsto a]}, [\![t]\!]_{\theta_2,\sigma_2[x \mapsto b]}) \in \Delta_{\tau_2,\rho}\,, \end{array}$$

so the induction hypothesis suffices.

In the case

$$\frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2},$$

we have

$$\begin{array}{ll} & ([\![t_1\ t_2]\!]_{\theta_1,\sigma_1}, [\![t_1\ t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_2,\rho} \\ \Leftrightarrow & ([\![t_1]\!]_{\theta_1,\sigma_1}\ \$\ [\![t_2]\!]_{\theta_1,\sigma_1}, [\![t_1]\!]_{\theta_2,\sigma_2}\ \$\ [\![t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_2,\rho} \\ \Leftarrow & ([\![t_2]\!]_{\theta_1,\sigma_1}, [\![t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}, \\ & \forall (a, b) \in \Delta_{\tau_1,\rho}.\ ([\![t_1]\!]_{\theta_1,\sigma_1}\ \$\ a, [\![t_1]\!]_{\theta_2,\sigma_2}\ \$\ b) \in \Delta_{\tau_2,\rho} \\ \Leftarrow & ([\![t_2]\!]_{\theta_1,\sigma_1}, [\![t_2]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}, \\ & ([\![t_1]\!]_{\theta_1,\sigma_1}, [\![t_1]\!]_{\theta_2,\sigma_2}) \in \Delta_{\tau_1 \to \tau_2,\rho}\,, \end{array}$$

so the induction hypotheses suffice.

Regarding

$$\frac{\alpha, \Gamma \vdash t :: \tau}{\Gamma \vdash (\Lambda\alpha.t) :: \forall\alpha.\tau},$$

we have

$$\begin{array}{ll} & ([\![\Lambda\alpha.t]\!]_{\theta_1,\sigma_1}, [\![\Lambda\alpha.t]\!]_{\theta_2,\sigma_2}) \in \Delta_{\forall\alpha.\tau,\rho} \\ \Leftrightarrow & (\lambda D_1.[\![t]\!]_{\theta_1[\alpha \mapsto D_1],\sigma_1}, \lambda D_2.[\![t]\!]_{\theta_2[\alpha \mapsto D_2],\sigma_2}) \in \Delta_{\forall\alpha.\tau,\rho} \\ \Leftrightarrow & \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in \mathit{Rel}(D_1, D_2). \\ & \quad ([\![t]\!]_{\theta_1[\alpha \mapsto D_1],\sigma_1}, [\![t]\!]_{\theta_2[\alpha \mapsto D_2],\sigma_2}) \in \Delta_{\tau,\rho[\alpha \mapsto \mathcal{R}]}\,, \end{array}$$

so the induction hypothesis suffices.

In the case

$$\frac{\Gamma \vdash t :: \forall \alpha.\tau_1}{\Gamma \vdash (t_{\tau_2}) :: \tau_1[\tau_2/\alpha]},$$

we have

$$\begin{array}{ll} & (\llbracket t_{\tau_2} \rrbracket_{\theta_1,\sigma_1}, \llbracket t_{\tau_2} \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau_1[\tau_2/\alpha],\rho} \\ \Leftrightarrow & (\llbracket t \rrbracket_{\theta_1,\sigma_1}\ \llbracket \tau_2 \rrbracket_{\theta_1}, \llbracket t \rrbracket_{\theta_2,\sigma_2}\ \llbracket \tau_2 \rrbracket_{\theta_2}) \in \Delta_{\tau_1,\rho[\alpha \mapsto \Delta_{\tau_2,\rho}]} \\ \Leftarrow & \forall D_1, D_2 \text{ pcpos}, \mathcal{R} \in Rel(D_1, D_2). \\ & \quad (\llbracket t \rrbracket_{\theta_1,\sigma_1}\ D_1, \llbracket t \rrbracket_{\theta_2,\sigma_2}\ D_2) \in \Delta_{\tau_1,\rho[\alpha \mapsto \mathcal{R}]} \\ \Leftrightarrow & (\llbracket t \rrbracket_{\theta_1,\sigma_1}, \llbracket t \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\forall \alpha.\tau_1,\rho}\,, \end{array}$$

so the induction hypothesis suffices. Note that the equivalence $\Delta_{\tau_1[\tau_2/\alpha],\rho} = \Delta_{\tau_1,\rho[\alpha \mapsto \Delta_{\tau_2,\rho}]}$, used in the first step above, holds by an easy induction on τ_1. Also note that the consecutive step uses $\Delta_{\tau_2,\rho} \in Rel$, as justified by Lemma 2.1.

In the case

$$\frac{\Gamma \vdash t :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ t :: \tau}$$

we have

$$\begin{array}{ll} & (\llbracket \mathbf{fix}\ t \rrbracket_{\theta_1,\sigma_1}, \llbracket \mathbf{fix}\ t \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau,\rho} \\ \Leftrightarrow & (\bigsqcup_{n\geq 0}\ (\llbracket t \rrbracket_{\theta_1,\sigma_1}\ \$)^n\ \bot, \bigsqcup_{n\geq 0}\ (\llbracket t \rrbracket_{\theta_2,\sigma_2}\ \$)^n\ \bot) \in \Delta_{\tau,\rho} \\ \Leftarrow & \forall n \in \mathbb{N}_0.(((\llbracket t \rrbracket_{\theta_1,\sigma_1}\ \$)^n\ (\bot), (\llbracket t \rrbracket_{\theta_2,\sigma_2}\ \$)^n\ (\bot)) \in \Delta_{\tau,\rho}) \\ \Leftarrow & \forall (x_1, x_2) \in \Delta_{\tau,\rho}.(((\llbracket t \rrbracket_{\theta_1,\sigma_1}\ \$)\ (x_1), (\llbracket t \rrbracket_{\theta_2,\sigma_2}\ \$)\ (x_2)) \in \Delta_{\tau,\rho}) \\ \Leftarrow & (\llbracket t \rrbracket_{\theta_1,\sigma_1}, \llbracket t \rrbracket_{\theta_2,\sigma_2}) \in \{(f,g) \mid f = \bot \text{ iff } g = \bot, \forall (a,b) \in \Delta_{\tau,\rho}.\ (f\ \$\ a, g\ \$\ b) \in \Delta_{\tau,\rho}\} \\ \Leftrightarrow & (\llbracket t \rrbracket_{\theta_1,\sigma_1}, \llbracket t \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau \to \tau,\rho} \end{array}$$

and therefore the premise suffices. Note that we used the strictness of $\Delta_{\tau,\rho}$ in the second, and additionally the continuity of $\Delta_{\tau,\rho}$ for the first implication, both given by Lemma 2.1.

Finally, in the case

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

we have to show that the values

$$\begin{cases} \llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x \mapsto a]} & \text{if } \llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = a \neq \bot \\ \bot & \text{if } \llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = \bot \end{cases}$$

and

$$\begin{cases} \llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x \mapsto b]} & \text{if } \llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = b \neq \bot \\ \bot & \text{if } \llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = \bot \end{cases}$$

are related by $\Delta_{\tau_2,\rho}$. By the induction hypothesis $(\llbracket t_1 \rrbracket_{\theta_1,\sigma_1}, \llbracket t_1 \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}$ and bottom-reflection of $\Delta_{\tau_1,\rho}$ (cf. Lemma 2.1) we only have to consider the following two cases:

1. $\llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = a \neq \bot$ and $\llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = b \neq \bot$, in which case the induction hypothesis that for every $(a,b) \in \Delta_{\tau_1,\rho}$,
$$(\llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x \mapsto a]}, \llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x \mapsto b]}) \in \Delta_{\tau_2,\rho}\,,$$
suffices, and

2. $\llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = \bot$ and $\llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = \bot$, in which case we have to show $(\bot,\bot) \in \Delta_{\tau_2,\rho}$, which follows from strictness of $\Delta_{\tau_2,\rho}$ (cf. Lemma 2.1 again).

This completes the proof.