Automatically Generating Counterexamples to Naive Free Theorems

Daniel Seidel* and Janis Voigtländer

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik
Römerstraße 164
53117 Bonn, Germany

{ds,jv}@iai.uni-bonn.de

Abstract. Disproof can be as important as proof in studying programs and programming languages. In particular, side conditions in a statement about program behavior are sometimes best understood and explored by trying to exhibit a falsifying example in the absence of a condition in question. Automation is as desirable for such falsification as it is for verification. We develop formal and implemented tools for counterexample generation in the context of free theorems, i.e., statements derived from polymorphic types à la relational parametricity. The machinery we use is rooted in constraining the type system and in intuitionistic proof search.

1 Introduction

Free theorems [19] as derived from relational parametricity [12] are an important source for useful statements about the semantics of programs in typed functional languages. But often, free theorems are derived in a "naive" setting, pretending that the language under consideration were conceptually as simple as the pure polymorphic lambda calculus [11] (maybe with algebraic data types added in) for which relational parametricity was originally conceived. For example, such a naive version claims that for every function, in Haskell syntax (no explicit "$\forall\alpha$."),

$$f :: [\alpha] \to [\alpha] \tag{1}$$

it holds that for every choice of types τ_1, τ_2, $g :: \tau_1 \to \tau_2$, and $x :: [\tau_1]$,

$$map\ g\ (f\ x) = f\ (map\ g\ x) \tag{2}$$

where $map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ is the standard function. But equivalence (2) does not actually hold in Haskell. A counterexample is obtained by setting

$$f = \lambda x \to [fix\ id], \quad \tau_1 = \tau_2 = \mathsf{Int}, \quad g = \lambda y \to 17, \quad x = [\,] \tag{3}$$

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

where *fix* :: $(\alpha \to \alpha) \to \alpha$ is a fixpoint combinator and *id* :: $\alpha \to \alpha$ the identity function. Indeed, now *map* g $(f\ x) = [17]$ but f $(map\ g\ x) = [\bot]$, where $\bot$ corresponds to nontermination. (Note that Haskell is lazy, so $[\bot] \neq \bot$.)

Trying to be less naive, we can take into account that the presence of fixpoint recursion enforces certain strictness conditions when deriving free theorems [19, Section 7] and thus obtain (2) only for g with $g\ \bot = \bot$. Generally, only sufficient, not always necessary, conditions are obtained. For example, the naive free theorem that for every function $h :: [\alpha] \to \mathsf{Int}$ it holds that $h\ x = h\ (map\ g\ x)$ becomes encumbered with the condition that, in the potential presence of *fix*, g should be strict. But here it is actually impossible to give a counterexample like (3) above. No matter how we complicate h by involving general recursion, in a pure lazy setting the naive free theorem holds steady, even for nonstrict g.

So it is natural to ask *when* it is the case that naive free theorems actually break in more realistic settings, and *how* to find corresponding counterexamples. Answering these questions will improve the understanding of free theorems and allow us to more systematically study corner cases in their applications. Indeed, concrete counterexamples were very important in studying semantic aspects of type-based program transformations in [8] and informed the invention of new transformations overcoming some of the encountered semantic problems [17]. (For a discussion of the general importance of counterexamples, see [9].) And also for other applications [16,18] it seems relevant to consider the potential negative impact of language extensions on free theorems. To date, counterexamples of interest have been literally *manu*factured, i.e., produced by hand in an ad-hoc fashion. With the work reported here we move to automatic generation instead, which is both an interesting problem and practically useful.

Summarily, the aim is as follows. Given a type, two versions of a free theorem can be produced: the naive one that ignores advanced language features and a more cautious one that comes with certain additional conditions. To explain whether, and if so why, a certain additional condition is really necessary, we want to provide a separating counterexample (or assert that there is none): a term of the given type such that the naive version of the free theorem fails for it precisely due to missing that condition.

Even when considering only the impact of general recursion and $\bot$ for now (as opposed to *seq* [7] or imprecise error semantics [15]), the set task is quite challenging; indeed much more so than the example of (2) vs. (3) suggests. Take, for example, the following type:

$$f :: ((\mathsf{Int} \to [\alpha]) \to \mathsf{Either\ Int\ Bool}) \to [\mathsf{Int}] \tag{4}$$

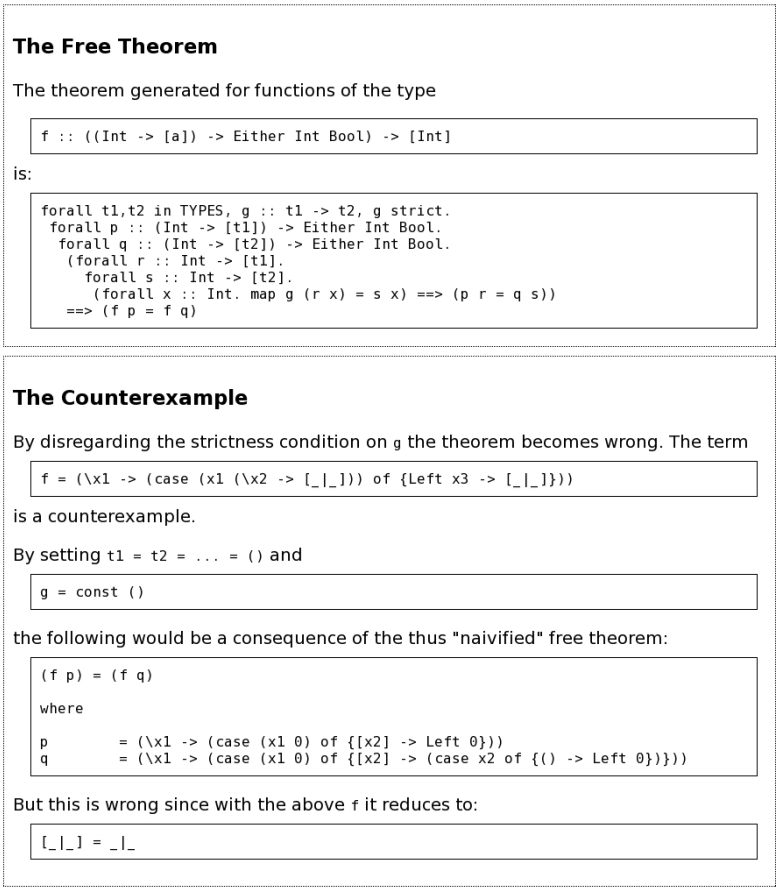
The "*fix*- and $\bot$-aware" free theorem generated for it is that for every choice of types τ_1, τ_2, strict function $g :: \tau_1 \to \tau_2$, and arbitrary functions $p :: (\mathsf{Int} \to [\tau_1]) \to \mathsf{Either\ Int\ Bool}$ and $q :: (\mathsf{Int} \to [\tau_2]) \to \mathsf{Either\ Int\ Bool}$,

$$\forall r :: \mathsf{Int} \to [\tau_1].\ p\ r = q\ (\lambda x \to map\ g\ (r\ x)) \tag{5}$$

implies

$$f\ p = f\ q \tag{6}$$

while the naive version would drop the strictness condition on g. The reader is invited to devise a concrete function term f of type (4) and concrete instantiations for τ_1, τ_2, (nonstrict) g, and p and q such that (5) holds but (6) fails. We have developed a system (and implemented it, and made it available online: `http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi`) which meets this challenge. A screenshot solving above puzzle is given below:



The Free Theorem

The theorem generated for functions of the type

```
f :: ((Int -> [a]) -> Either Int Bool) -> [Int]
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
 forall p :: (Int -> [t1]) -> Either Int Bool.
  forall q :: (Int -> [t2]) -> Either Int Bool.
   (forall r :: Int -> [t1].
      forall s :: Int -> [t2].
       (forall x :: Int. map g (r x) = s x) ==> (p r = q s))
   ==> (f p = f q)
```

The Counterexample

By disregarding the strictness condition on `g` the theorem becomes wrong. The term

```
f = (\x1 -> (case (x1 (\x2 -> [_|_])) of {Left x3 -> [_|_]}))
```

is a counterexample.

By setting `t1 = t2 = ... = ()` and

```
g = const ()
```

the following would be a consequence of the thus "naivified" free theorem:

```
(f p) = (f q)

where

p        = (\x1 -> (case (x1 0) of {[x2] -> Left 0}))
q        = (\x1 -> (case (x1 0) of {[x2] -> (case x2 of {() -> Left 0})}))
```

But this is wrong since with the above `f` it reduces to:

```
[_|_] = _|_
```

Since a counterexample consists of several terms that need to fit together in a very specific way, also guaranteeing extra conditions like the relationship (5) between p, q, and g in the case just considered, a random or unguided exhaustive search approach would be unsuitable here. That is, successfully using a tool in the spirit of QuickCheck [3] to refute naive free theorems by finding counterexamples would require extremely smart and elaborate generators to prevent a "needle in a haystack search". Indeed, we contend that the required generators would have to be so complex and ad-hoc, and extra effort would have to go into enforcing a suitable search strategy, that there would be no benefit left from using a generic testing framework. We should hence instead directly go for a formal algorithm. (We did, however, use QuickCheck to validate our implemented generator.)

The approach we do follow is based on first capturing what is *not* a counterexample. Even in a language including general recursion and $\bot$ there will be terms that do not involve either of them or that do so but in a way not affecting a given naive free theorem. It is possible to develop a refined type system that keeps track of uses of *fix* (and thus $\bot$) and admits a refined notion of relational parametricity in which fewer strictness conditions are imposed, depending on the recorded information. This refinement allows us to describe a subset of the terms of a given (original) type for which a certain strictness condition in the "*fix*-aware" free theorem can actually be dropped. This idea was pioneered in [10], and we provide an equivalent formalization tailored to our purposes.

Knowing how to describe what is not a counterexample, we can then systematically search for a term that is *not* not a counterexample. That is, we look for a term that has the original type but *not* the refined type (in the refined type system). For the term search we take inspiration from a decision procedure for intuitionistic propositional logic in [5]. This procedure can be turned into a *fix*-free term generator for polymorphic types, and was indeed put to use so [1]. Our twist is that we do allow the generation of terms containing *fix*, but not arbitrarily. Allowing arbitrary occurrences would lead to a trivial generator, because *fix id* can be given any type. So instead we design the search in such a way that at least one use of *fix* is ensured in a position where it enforces a restriction in the refined type system, and truly separates between "harmful/harmless encounter of $\bot$". Thus we really find a term (if one exists) which is in the difference of the set of all terms and the subset containing only the "not a counterexample" terms.

At this point we have, in the terminology of [9], found a *local* counterexample: a term for which the refined type system and its refined notion of relational parametricity cannot prove that a certain strictness condition can be dropped. What we really want is a *global* counterexample: a term for which there cannot be *any* proof that the strictness condition in question can be dropped. Turning a local counterexample into a global one requires additional work, in particular to come up with appropriate instantiations like for τ_1, τ_2, g, p, and q in the challenge regarding (4) above. We return to this issue, also based on example (1), in Section 6. It turns out that not all our local counterexamples can be turned into global ones using our proposed construction, but where that construction succeeds we have a correctness statement (proved in [13]); correctness meaning that any counterexample we offer really contradicts the naive free theorem in question. Moreover, we claim completeness; meaning that if our method *finds* no *local* counterexample, then there *is* no *global* counterexample.

2 The Calculus and Standard Parametricity

We set out from a standard denotational semantics for a polymorphic lambda calculus, called **PolyFix**, that corresponds to a core of Haskell (without *seq*, without type classes, without special treatment of errors or exceptions, ...).

Types are formed according to the following grammar, where α ranges over type variables: $\tau ::= \alpha \mid \tau \to \tau \mid [\tau] \mid (\tau, \tau) \mid \mathsf{Either}\ \tau\ \tau \mid ()$. We include lists,

pairs, and a disjoint sum type as representatives for algebraic data types. Additionally, we include a unit type () to be used later on. (And our implementation additionally deals with Int, Bool, Maybe.) Note that there is no case $\forall\alpha.\tau$ in the grammar, because we will only consider rank-1 polymorphism (as in Haskell 98). Dealing with higher-rank polymorphism, and thus local quantification over type variables, would complicate our technique immensely.

Terms are formed according to the grammar $t ::= x \mid \lambda x :: \tau.t \mid t\ t \mid []_\tau \mid t : t \mid (t,t) \mid \mathsf{Left}_\tau\ t \mid \mathsf{Right}_\tau\ t \mid () \mid \mathbf{fix}\ t \mid \mathbf{case}\ t\ \mathbf{of}\ \{\cdots\}$, where x ranges over term variables. There are versions of $\mathbf{case}\ t\ \mathbf{of}\ \{\cdots\}$ for lists, pairs, disjoint sum types, and the unit type (each with the obvious configuration of exhaustive pattern match branches). Terms are typed according to standard typing rules. We give only some examples in Fig. 1. Type and term variable contexts Γ and Σ are unordered sets of the forms $\alpha_1, \ldots, \alpha_k$ and $x_1 :: \tau_1, \ldots, x_l :: \tau_l$.

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

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

$$\frac{\Gamma; \Sigma \vdash t_1 :: \tau_1 \qquad \Gamma; \Sigma \vdash t_2 :: \tau_2}{\Gamma; \Sigma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ (\textsc{Pair}) \qquad \frac{\Gamma; \Sigma \vdash t :: \tau_1}{\Gamma; \Sigma \vdash (\mathsf{Left}_{\tau_2}\ t) :: \mathsf{Either}\ \tau_1\ \tau_2}\ (\textsc{Left})$$

$$\frac{\Gamma; \Sigma \vdash t :: \tau \to \tau}{\Gamma; \Sigma \vdash (\mathbf{fix}\ t) :: \tau}\ (\textsc{Fix})$$

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

Fig. 1. Some of the Typing Rules for **PolyFix**.

The denotational semantics interprets types as *pointed complete partial orders* (least element always denoted $\bot$), functions as monotonic and continuous, but not necessarily strict, maps (ordered point-wise), and is entirely standard. Of note is that the interpretations of pair and disjoint sum types are noncoalesced:

$$\begin{aligned}
[\![(\tau_1, \tau_2)]\!]_\theta &= \mathit{lift}_\bot\,\{(a,b) \mid a \in [\![\tau_1]\!]_\theta,\, b \in [\![\tau_2]\!]_\theta\} \\
[\![\mathsf{Either}\ \tau_1\ \tau_2]\!]_\theta &= \mathit{lift}_\bot(\{\mathsf{Left}\ a \mid a \in [\![\tau_1]\!]_\theta\} \cup \{\mathsf{Right}\ a \mid a \in [\![\tau_2]\!]_\theta\})
\end{aligned}$$

The operation $\mathit{lift}_\bot$ takes a complete partial order, adds a new element $\bot$ to the carrier set, and defines this new $\bot$ to be below every other element. For the semantics of terms we also show just a few example cases:

$$[\![x]\!]_\sigma = \sigma(x), \quad [\![\lambda x :: \tau.t]\!]_\sigma\ a = [\![t]\!]_{\sigma\{x \mapsto a\}}, \quad [\![t_1\ t_2]\!]_\sigma = [\![t_1]\!]_\sigma\ [\![t_2]\!]_\sigma, \quad [\![()]\!]_\sigma = (),$$

$$[\![\mathbf{case}\ t\ \mathbf{of}\ \{() \to t_1\}]\!]_\sigma\ =\ \begin{cases} [\![t_1]\!]_\sigma & \text{if } [\![t]\!]_\sigma = () \\ \bot & \text{if } [\![t]\!]_\sigma = \bot \end{cases}$$

Altogether, we have that if $\Gamma; \Sigma \vdash t :: \tau$ and $\sigma(x) \in [\![\tau']\!]_\theta$ for every $x :: \tau'$ occurring in Σ, then $[\![t]\!]_\sigma \in [\![\tau]\!]_\theta$.

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 as follows, assuming ρ to be a mapping from type variables to binary relations between complete partial orders:

$$\begin{aligned}
\Delta_{\alpha,\rho} &= \rho(\alpha) \\
\Delta_{\tau_1\to\tau_2,\rho} &= \{(f,g) \mid \forall (a,b) \in \Delta_{\tau_1,\rho}.\ (f\ a, g\ b) \in \Delta_{\tau_2,\rho}\} \\
\Delta_{[\tau],\rho} &= \mathit{lfp}\,(\lambda\mathcal{S}.\{(\bot,\bot),\ ([\,],[\,])\} \\
&\qquad \cup \{(a:b, c:d) \mid (a,c) \in \Delta_{\tau,\rho},\ (b,d) \in \mathcal{S}\}) \\
\Delta_{(\tau_1,\tau_2),\rho} &= \{(\bot,\bot)\} \cup \{((a,b),(c,d)) \mid (a,c) \in \Delta_{\tau_1,\rho},\ (b,d) \in \Delta_{\tau_2,\rho}\} \\
\Delta_{\mathsf{Either}\ \tau_1\ \tau_2,\rho} &= \{(\bot,\bot)\} \cup \{(\mathsf{Left}\ a, \mathsf{Left}\ b) \mid (a,b) \in \Delta_{\tau_1,\rho}\} \cup \{\cdots\} \\
\Delta_{(),\rho} &= id_{\{\bot,()\}}
\end{aligned}$$

For two pointed complete partial orders D_1 and D_2, let $Rel^{\bot}(D_1, D_2)$ collect all relations between them that are *strict* (i.e., contain the pair $(\bot,\bot)$) and *continuous* (i.e., are closed under suprema). Also, let $Rel^{\bot}$ be the union of all $Rel^{\bot}(D_1, D_2)$. The following parametricity theorem is standard [12,19].

Theorem 1. *If $\Gamma; \Sigma \vdash t :: \tau$, then $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \in \Delta_{\tau,\rho}$ for every θ_1, θ_2, ρ, σ_1, and σ_2 such that for every α occurring in Γ, $\rho(\alpha) \in Rel^{\bot}(\theta_1(\alpha), \theta_2(\alpha))$, and for every $x :: \tau'$ occurring in Σ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$.*

An important special case in practice is when ρ maps only to functions.

3 Refining the Type System to Put fix under Control

The requirement $\rho(\alpha) \in Rel^{\bot}$ in Theorem 1 is responsible for strictness conditions like those on the g in the examples in the introduction. It is required because due to **fix** some parts of the theorem's proof really depend on the strictness of relational interpretations of (certain) types. Launchbury and Paterson [10] proposed to explicitly keep track, in the type of a term, of uses of general recursion and/or $\bot$. One of their aims was to provide less restrictive free theorems for situations where **fix** is known not to be used in a harmful way. While they formulated their ideas using Haskell's type classes, a direct formalization in typing rules is possible as well.

In a type variable context Γ we now distinguish between variables on which general recursion is not allowed and those on which it is. We simply annotate the latter type variables by *. The idea is that a derivation step

$$\frac{\Gamma; \Sigma \vdash t :: \alpha \to \alpha}{\Gamma; \Sigma \vdash (\mathbf{fix}\ t) :: \alpha}$$

is only allowed if α is thus annotated in Γ. Since the actual rule (Fix) mentions an arbitrary τ, rather than more specifically a type variable, we need propagation rules describing when a type supports the use of **fix**. This need is addressed by defining a predicate $\mathsf{Pointed}$ on types. The base and the sole propagation rule are:

$$\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Pointed}} \qquad \frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed}}{\Gamma \vdash (\tau_1 \to \tau_2) \in \mathsf{Pointed}}$$

In addition, axioms assert that algebraic data types support **fix**: $\Gamma \vdash () \in \mathsf{Pointed}$, $\Gamma \vdash [\tau] \in \mathsf{Pointed}$, $\Gamma \vdash (\tau_1, \tau_2) \in \mathsf{Pointed}$, $\Gamma \vdash (\mathsf{Either}\ \tau_1\ \tau_2) \in \mathsf{Pointed}$.

To the typing rules (Fix) and (Case) from Fig. 1 we can now add the premise $\Gamma \vdash \tau \in \mathsf{Pointed}$, and similarly to the typing rules for the versions of **case** t **of** $\{\cdots\}$ for pairs, disjoint sum types, and the unit type. The resulting system is called **PolyFix***. Note that the syntax of types and terms is the same in **PolyFix** and in **PolyFix***. But depending on which type variables are *-annotated in Γ, the latter may have fewer typable terms at a given type. The denotational semantics remains as before, except that for $\Gamma; \Sigma \vdash t :: \tau$ we can choose to map non-* type variables in Γ to just *complete partial orders* (rather than necessarily to *pointed* ones) in the type environment θ.

The benefit of recording at which types **fix** may be used is that we can now give a parametricity theorem with relaxed preconditions. For two complete partial orders D_1 and D_2, let $Rel(D_1, D_2)$ collect all relations between them that are continuous (but not necessarily strict). For the very same logical relation Δ as in Section 2 we then have the following variant of Theorem 1, proved in [13].

Theorem 2. *If $\Gamma; \Sigma \vdash t :: \tau$ in **PolyFix***, then $(\llbracket t \rrbracket_{\sigma_1}, \llbracket t \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}$ for every θ_1, θ_2, ρ, σ_1, and σ_2 such that for every α occurring in Γ, $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$, for every α^* occurring in Γ, $\rho(\alpha) \in Rel^{\perp}(\theta_1(\alpha), \theta_2(\alpha))$, and for every $x :: \tau'$ occurring in Σ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$.*

This brings us on a par with [10]. While Launchbury and Paterson stopped at this point, being able to determine for a specific term that its use (or non-use) of **fix** and $\perp$ does not require certain strictness conditions that would have to be imposed when just knowing the term's type, we instead use this result as a stepping stone. Our aim is somewhat inverse to theirs: we will start from just a type and try to find a specific term using **fix** in such a way that a certain side condition *is* required.

4 Term Search, Motivation

Theorem 2 tells us that sometimes strictness conditions are not required in free theorems. For example, we now know that for every function f typable as $\alpha \vdash f :: [\alpha] \to [\alpha]$ (i.e., with non-* α) in **PolyFix*** strictness of g is not required for (2) from the introduction to hold. Correspondingly, the function $f = \lambda x \to [\mathit{fix}\ \mathit{id}]$ (or rather, $f = \lambda x :: [\alpha].(\mathbf{fix}\ (\lambda y :: \alpha.y)) : [\,]_\alpha$) used for the counterexample in (3) is not so typable. It is only typable as $\alpha^* \vdash f :: [\alpha] \to [\alpha]$ in **PolyFix***. This observation agrees with our general strategy for finding counterexamples as already outlined in the introduction.

Given a polymorphic type signature containing one or more type variables, and a free theorem derived from it that contains one or more strictness conditions due to the $\rho(\alpha) \in Rel^{\perp}$, $\rho(\beta) \in Rel^{\perp}$, ... from Theorem 1, assume that we want to explain (or refute) the necessity of one particular among these strictness conditions, say of the one originating from $\rho(\alpha) \in Rel^{\perp}$, and that we want to do so by investigating the existence of a counterexample to the more naive

free theorem obtained by dropping that particular strictness condition. This investigation can now be done by searching a term that is typable with the given signature in **PolyFix*** under context $\alpha^*, \beta^*, \ldots$, but not under $\alpha, \beta^*, \ldots$.

Unfortunately, this term search cannot use the typing rules of **PolyFix** and/or **PolyFix*** themselves, because in proof theory terminology they lack the subformula property. For example, rule (App) from Fig. 1 with terms (and Γ) omitted corresponds to modus ponens. Reading it upwards we have to invent τ_1 without any guidance being given by τ_2. Rule (Case) is similarly problematic. In proof search for intuitionistic propositional logic this problem is solved by designing rule systems that have the same proof power but additionally *do* enjoy the subformula property in the sense that one can work out what formulas can appear in a proof of a particular goal sequent. One such system is that of [5]. It can be extended to inductive constructions and then turned into a term generator for "**PolyFix** $\setminus \{$**fix**$\}$". This extension serves as starting point for our search of **PolyFix*** terms typable under α^* but not under just α, in the next section.

Specifically, terms typable to a given type in "**PolyFix** $\setminus \{$**fix**$\}$" can be generated based on a system extending that of [5] by rules for inductive definitions (to deal with lists, pairs, ...) à la those of [4]. The resulting system retains some of the rules from **PolyFix** (e.g., (Var), (Nil), (Unit), (Abs), (Pair), and (Left) from Fig. 1), naturally drops (Fix), replaces (App) by[1]

$$\frac{\Gamma; \Sigma, x :: \tau_1, y :: \tau_2 \vdash t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \to \tau_2, x :: \tau_1 \vdash [y \mapsto f\ x]t :: \tau}\ (\text{App'})$$

adds the rule (Arrow$\to$):

$$\frac{\Gamma; \Sigma, x :: \tau_1, g :: \tau_2 \to \tau_3 \vdash t_1 :: \tau_2 \qquad \Gamma; \Sigma, y :: \tau_3 \vdash t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \to \tau_2) \to \tau_3 \vdash [y \mapsto f\ (\lambda x :: \tau_1.[g \mapsto \lambda z :: \tau_2.f\ (\lambda u :: \tau_1.z)]t_1)]t_2 :: \tau}$$

and further rules for dealing with data types. We do not go into more detail here (but see Section 4 of [13]), because we will anyway discuss the full rule system for counterexample term search in **PolyFix*** in the next section.

5 Term Search, The Algorithm

The prime way to provoke a term to be typable in **PolyFix*** under context $\alpha^*, \ldots$ but not under context $\alpha, \ldots$ is to use **fix** at a type whose membership in $\mathsf{Pointed}$ depends on α being *-annotated.[2] So a natural first rule for our new system **TermFind** is the following one:

$$\frac{\Gamma \vdash \tau \notin \mathsf{Pointed}}{\Gamma; \Sigma \Vdash \bot_\tau :: \tau}\ (\text{Bottom})$$

[1] We use the notation $[y \mapsto \cdots]t$ for substituting y in t by "$\cdots$".

[2] The typing rules for **case** t **of** $\{\cdots\}$ in **PolyFix*** also each have a "$\in \mathsf{Pointed}$" precondition, but those terms do not introduce any $\bot$ values by themselves. Only if the evaluation of t is already $\bot$, the overall term would evaluate to $\bot$. Thus, every occurrence of $\bot$ (to which all trouble with naive free theorems in our setting eventually boils down) originates from a use of **fix**.

where we use the syntactic abbreviation $\bot_\tau = \mathbf{fix}\ (\lambda x :: \tau.x)$. Note the new symbol $\Vdash$, instead of $\vdash$, for term level rules. The rules defining the predicate $\mathsf{Pointed}$ on types are kept unchanged, and $\Gamma \vdash \tau \notin \mathsf{Pointed}$ simply means that $\Gamma \vdash \tau \in \mathsf{Pointed}$ is not derivable.

In a judgment of the form $\Gamma; \Sigma \Vdash t :: \tau$, the type and term variable contexts Γ and Σ, as well as the type τ, are considered as "inputs", while the term t is taken to be the produced "output". If $\alpha, \ldots \vdash \tau \notin \mathsf{Pointed}$ but $\alpha^*, \ldots \vdash \tau \in \mathsf{Pointed}$, then with an $\alpha, \ldots; \Sigma \Vdash \bot_\tau :: \tau$ obtained according to the above rule we have really found a term in the intended difference set. Of course, we will not always be so lucky that the type for which we originally want to find a counterexample term is itself one whose "pointedness" depends on an α of interest. So in general we have to do a real search for an opportunity to "inject" a harmful $\bot$.

For example, if the type for which we are searching for a counterexample is $\mathsf{Either}\ \tau_1\ \tau_2$, we could try to find a counterexample term for τ_1 and then build the one for $\mathsf{Either}\ \tau_1\ \tau_2$ from it via Left_{τ_2}. That is, **TermFind** takes over (modulo replacing $\vdash$ by $\Vdash$) rule (Left) from Fig. 1 and also the corresponding rule (Right). At list types, we can search at the element type level, then wrap the result in a singleton list:

$$\frac{\Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma \Vdash (t : [\,]_\tau) :: [\tau]}\ (\textsc{Wrap})$$

For pair types we have a choice similarly to (Left) vs. (Right) above. The other pair component in each case is simply filled with an appropriate $\bot$-term:[3]

$$\frac{\Gamma; \Sigma \Vdash t :: \tau_1}{\Gamma; \Sigma \Vdash (t, \bot_{\tau_2}) :: (\tau_1, \tau_2)}\ (\textsc{Pair}_1) \qquad \frac{\Gamma; \Sigma \Vdash t :: \tau_2}{\Gamma; \Sigma \Vdash (\bot_{\tau_1}, t) :: (\tau_1, \tau_2)}\ (\textsc{Pair}_2)$$

For the unit type, there is no hope of introducing an "α is *-annotated"-enforcing $\bot$ directly, i.e., without using material from the term variable context Σ. For function types we can *bring* material into the term variable context by using the rule (Abs), with $\Vdash$ instead of $\vdash$, from Fig. 1. Material that once *is* in the context may or may not be useful for eventually constructing a counterexample. But in certain cases we can simplify it without danger of missing out on some possible counterexample. For example, if we have a pair (variable) in the context, then we can be sure that if a counterexample term can be found using it, the same would be true if we replaced the pair by its components. After all, from any such counterexample using the components one could also produce a counterexample based on the pair itself, involving simple projections. Hence:

$$\frac{\Gamma; \Sigma, x :: \tau_1, y :: \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, p :: (\tau_1, \tau_2) \Vdash [x \mapsto \mathbf{fst}\ p, y \mapsto \mathbf{snd}\ p]t :: \tau}\ (\textsc{Proj})$$

where we use abbreviations $\mathbf{fst}\ p$ and $\mathbf{snd}\ p$, e.g., $\mathbf{fst}\ p = \mathbf{case}\ p\ \mathbf{of}\ \{(x, y) \to x\}$.

[3] Intuitively, once we have found a counterexample term $t :: \tau_1$, we can as well add $\bot$ in other places. Note that using, say, $\bot_{\tau_2}$ does not require us to put a $\Gamma \vdash \tau_2 \in \mathsf{Pointed}$ constraint in. After all, we will require typability of the resulting term in **PolyFix*** only under the context with all type variables *-annotated anyway.

Similarly, we obtain the following rule:

$$\frac{\Gamma; \Sigma, h :: \tau_1 \Vdash t :: \tau}{\Gamma; \Sigma, l :: [\tau_1] \Vdash [h \mapsto \mathbf{head}_{\tau_1}\ l]t :: \tau}\ (\textsc{Head})$$

where we use the abbreviation $\mathbf{head}_{\tau_1}\ l = \mathbf{case}\ l\ \mathbf{of}\ \{[\,] \to \bot_{\tau_1}\,;\ x : y \to x\}$. (As with rule (Wrap), we only ever use lists via their first elements.)

For a type $\mathsf{Either}\ \tau_1\ \tau_2$ in the context, since we do not know up front whether we will have more success constructing a counterexample when replacing it with τ_1 or with τ_2, we get two rules that are in competition with each other. One of them (the other one, ($\textsc{Dist}_2$), is analogous) looks as follows:

$$\frac{\Gamma; \Sigma, x :: \tau_1 \Vdash t :: \tau}{\Gamma; \Sigma, e :: \mathsf{Either}\ \tau_1\ \tau_2 \Vdash [x \mapsto \mathbf{fromLeft}_{\tau_1}\ e]t :: \tau}\ (\textsc{Dist}_1)$$

where we abbreviate $\mathbf{fromLeft}_{\tau_1}\ e = \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Left}\ x \to x\,;\ \mathsf{Right}\ x \to \bot_{\tau_1}\}$.

Unit and unpointed types in the context are of no use for counterexample generation, because no relevant α-affecting $\bot$ can be "hidden" in them. The following two rules reflecting this observation do not really contribute to the discovery of a solution term, but can shorten the search process by removing material that then needs not to be considered anymore.

$$\frac{\Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma, x :: () \Vdash t :: \tau}\ (\textsc{Drop}_1) \qquad \frac{\Gamma \vdash \tau_1 \notin \mathsf{Pointed} \qquad \Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma, x :: \tau_1 \Vdash t :: \tau}\ (\textsc{Drop}_2)$$

What remains to be done is to deal with (pointed) function types in the context. We distinguish several cases according to what is on the input side of those function types. The intuitive idea is that an input of an input corresponds to an output (akin to a logical double negation translation). Thus, for example, the following rule corresponds to the rule (Wrap) seen earlier:

$$\frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed} \qquad \Gamma; \Sigma, g :: \tau_1 \to \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, f :: [\tau_1] \to \tau_2 \Vdash [g \mapsto \lambda x :: \tau_1.f\ (x : [\,]_{\tau_1})]t :: \tau}\ (\textsc{Wrap}{\to})$$

For pairs we introduce a rule corresponding to currying:

$$\frac{\Gamma \vdash \tau_3 \in \mathsf{Pointed} \qquad \Gamma; \Sigma, g :: \tau_1 \to \tau_2 \to \tau_3 \Vdash t :: \tau}{\Gamma; \Sigma, f :: (\tau_1, \tau_2) \to \tau_3 \Vdash [g \mapsto \lambda x :: \tau_1.\lambda y :: \tau_2.f\ (x, y)]t :: \tau}\ (\textsc{Pair}{\to})$$

Similarly, we introduce the following rule ($\textsc{Either}{\to}$):

$$\frac{\Gamma \vdash \tau_3 \in \mathsf{Pointed} \qquad \Gamma; \Sigma, g :: \tau_1 \to \tau_3, h :: \tau_2 \to \tau_3 \Vdash t :: \tau}{\begin{array}{c}\Gamma; \Sigma, f :: \mathsf{Either}\ \tau_1\ \tau_2 \to \tau_3 \\ \Vdash \\ [g \mapsto \lambda x :: \tau_1.f\ (\mathsf{Left}_{\tau_2}\ x), h \mapsto \lambda x :: \tau_2.f\ (\mathsf{Right}_{\tau_1}\ x)]t :: \tau\end{array}}$$

Independently of the input side of a function type in the context we can always simplify such a type by providing a dummy $\bot$-term:

$$\frac{\Gamma; \Sigma, x :: \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \to \tau_2 \Vdash [x \mapsto f\ \bot_{\tau_1}]t :: \tau}\ (\textsc{Bottom}{\to})$$

This $\bot$ by itself will not generally contribute to the overall constructed term being a counterexample, but can enable important progress in the search.

A variant of (Bottom$\to$) that would be more directly promising for injecting a harmful $\bot$ would be if we demanded $\Gamma \vdash \tau_1 \notin \mathsf{Pointed}$ as a precondition. The resulting rule would correspond to rule (Bottom) from the beginning of this section. However, here additional effort is needed in order to ensure that a t of type τ generated from context $\Gamma; \Sigma, x :: \tau_2$ can really lead to the term $[x \mapsto f\ \bot_{\tau_1}]t$ being a counterexample (of type τ, in context $\Gamma; \Sigma, f :: \tau_1 \to \tau_2$). Namely, we need to ensure that x really occurs in t, and is actually used in a somehow "essential" way, because otherwise the $\bot_{\tau_1}$ we inject would be for naught, and could not provoke a breach of the naive free theorem under consideration. This notion of "essential use" (related to relevance typing [6,20]) will be formalized by a separate rule system $\Vdash^\circ$ below. Using it, our rule variant becomes:

$$\frac{\Gamma \vdash \tau_1 \notin \mathsf{Pointed} \qquad \Gamma; \Sigma, x^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \to \tau_2 \Vdash [x \mapsto f\ \bot_{\tau_1}]t :: \tau}\ (\text{Bottom}\to\text{'})$$

Of the rule (Arrow$\to$) mentioned at the end of Section 4, we also give a variant, called (Arrow$\to^\circ$), that employs $\Vdash^\circ$ to enforce an essential use. It is:

$$\frac{\Gamma \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \quad \Gamma; \Sigma, x :: \tau_1, g :: \tau_2 \to \tau_3 \Vdash t_1 :: \tau_2 \quad \Gamma; \Sigma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash [y \mapsto f\ (\lambda x :: \tau_1.[g \mapsto \lambda z :: \tau_2.f\ (\lambda u :: \tau_1.z)]t_1)]t_2 :: \tau}$$

If $\Gamma \vdash \tau_2 \notin \mathsf{Pointed}$, then the use of (Bottom$\to$') will promise more immediate success, by requiring only (an equivalent of) the last precondition.

The purpose of rule system $\Vdash^\circ$ is to produce, given a pair of type and term variable contexts in which some term variables may be $^\circ$-annotated, a term of a given type such that an evaluation of that term is not possible without accessing the value of at least one of the term variables that are $^\circ$-annotated in the context. The simplest rule naturally is as follows:

$$\Gamma; \Sigma, x^\circ :: \tau \Vdash^\circ x :: \tau\ (\text{Var}^\circ)$$

Other rules are by analyzing the type of some $^\circ$-annotated term variable in the context. For example, if we find a so annotated variable of a pair type, then an essential use of that variable can be enforced by enforcing the use of either of its components. This observation leads to the following variant of rule (Proj):

$$\frac{\Gamma; \Sigma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, p^\circ :: (\tau_1, \tau_2) \Vdash^\circ [x \mapsto \mathbf{fst}\ p, y \mapsto \mathbf{snd}\ p]t :: \tau}\ (\text{Proj}^\circ)$$

Analogously, $^\circ$-variants of the rules (Head), (Dist$_1$), and (Dist$_2$) are obtained.

The only way to enforce the use of a variable of function type is to apply it to some argument and enforce the use of the result. The argument itself is irrelevant for doing so, hence we get the following variant of rule (Bottom$\to$):

$$\frac{\Gamma; \Sigma, x^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f^\circ :: \tau_1 \to \tau_2 \Vdash^\circ [x \mapsto f\ \bot_{\tau_1}]t :: \tau}\ (\text{Bottom}\to^\circ)$$

Another possibility to use a $^\circ$-annotated term variable is to provide it as argument to another term variable that has a function type with matching input side. That term variable of function type needs not itself be $^\circ$-annotated. In fact, if it were, it could already have been eliminated by (Bottom$\to^\circ$). But it is essential to enforce that the function result is used in the overall term if the argument is not already used via other means. Thus, we get as variant of (App'):

$$\frac{\Gamma \vdash \tau_2 \in \mathsf{Pointed} \qquad \Gamma; \Sigma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \to \tau_2, x^\circ :: \tau_1 \Vdash^\circ [y \mapsto f\ x]t :: \tau}\ (\text{App'}^\circ)$$

Finally, we can try to use a $^\circ$-annotated term variable as scrutinee for **case**. If we have a $^\circ$-annotated term variable of unit type, we need axioms of the form

$$\Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\mathbf{case}\ x\ \mathbf{of}\ \{() \to t\}) :: \tau$$

where we just have to guarantee that t has type τ in context $\Gamma; \Sigma$ and that it does not evaluate to $\bot$. Since the "first phase" $\Vdash$ has already done most of the work of deconstructing types, the following axioms suffice:

$$\Gamma; \Sigma, y :: \tau, x^\circ :: () \Vdash^\circ (\mathbf{case}\ x\ \mathbf{of}\ \{() \to y\}) :: \tau\ (\text{Unit}^\circ\text{–Var'})$$
$$\Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\mathbf{case}\ x\ \mathbf{of}\ \{() \to ()\}) :: ()\ (\text{Unit}^\circ\text{–Unit'})$$
$$\Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\mathbf{case}\ x\ \mathbf{of}\ \{() \to [\bot_\tau]\}) :: [\tau]\ (\text{Unit}^\circ\text{–List'})$$

plus similar (Unit$^\circ$–Pair') and (Unit$^\circ$–Either'). Note that in (Unit$^\circ$–Var'), using y is okay, since we can find environments in which its evaluation is non-$\bot$. Axioms analogous to the ones just considered are added for every other possible type of x supporting **case**, e.g.,

$$\Gamma; \Sigma, y :: \tau, x^\circ :: [\tau_1] \Vdash^\circ (\mathbf{case}\ x\ \mathbf{of}\ \{[z] \to y\}) :: \tau\ (\text{List}^\circ\text{–Var'})$$

So far we have not set up an explicit order in which the rules of $\Vdash$ and $\Vdash^\circ$ should be applied. In fact, they could be used in arbitrary order, and using an appropriate measure (based on the structure and nesting levels of types) we have shown that even full backtracking search terminates [13]. That is, starting with a triple of Γ (potentially containing *-annotated type variables), Σ, and τ, either a term t with $\Gamma; \Sigma \Vdash t :: \tau$ is produced, or it is guaranteed that there is no such term. In the implementation, we actually use a safely reduced amount of backtracking and an optimized order, both based on ideas from [4]. The full rule systems, incorporating order and backtracking information, are given in [13].

Example. For "$\alpha; \Vdash t :: ([\alpha] \to ()) \to ()$" as target judgement, the term $t = \lambda f :: [\alpha] \to ().(\lambda x :: \alpha.f\ (x : [\,]_\alpha))\ \bot_\alpha$ is found as follows:

$$\dfrac{\dfrac{\dfrac{\alpha \vdash \alpha \notin \mathsf{Pointed} \quad \alpha; x^\circ :: () \Vdash^\circ x :: ()\ (\text{Var}^\circ)}{\alpha; g :: \alpha \to () \Vdash (g\ \bot_\alpha) :: ()}\ (\text{Bottom}\to\text{'}) \quad \alpha \vdash () \in \mathsf{Pointed}}{\alpha; f :: [\alpha] \to () \Vdash ((\lambda x :: \alpha.f\ (x : [\,]_\alpha))\ \bot_\alpha) :: ()}\ (\text{Wrap}\to)}{\alpha; \Vdash (\lambda f :: [\alpha] \to ().(\lambda x :: \alpha.f\ (x : [\,]_\alpha))\ \bot_\alpha) :: ([\alpha] \to ()) \to ()}\ (\text{Abs})$$

This t is such that $\alpha^*;\vdash t :: ([\alpha] \to ()) \to ()$ holds in **PolyFix***, while $\alpha;\vdash t :: ([\alpha] \to ()) \to ()$ does not. Indeed, for $\Gamma = \alpha^*$ we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \mathsf{Pointed}} \quad \dfrac{\Gamma; f :: [\alpha] \to (), x :: \alpha \vdash x :: \alpha}{\Gamma; f :: [\alpha] \to () \vdash (\lambda x :: \alpha.x) :: \alpha \to \alpha}\,(\textsc{Abs})}{\Gamma; f :: [\alpha] \to () \vdash \bot_\alpha :: \alpha}\,(\textsc{Fix})}{\vdots}}{\Gamma; f :: [\alpha] \to () \vdash ((\lambda x :: \alpha.f\ (x : [\,]_\alpha))\ \bot_\alpha) :: ()}\,(\textsc{App})}{\Gamma;\vdash (\lambda f :: [\alpha] \to ().(\lambda x :: \alpha.f\ (x : [\,]_\alpha))\ \bot_\alpha) :: ([\alpha] \to ()) \to ()}\,(\textsc{Abs})$$

while for $\Gamma = \alpha$ there would be no successful typing.

In general, we get:

Theorem 3. *Let (Γ, Σ, τ) be an input for the term search. Let Γ^* be as Γ, but all type variables *-annotated. If term search returns some t, i.e., if $\Gamma; \Sigma \Vdash t :: \tau$, then in **PolyFix*** $\Gamma; \Sigma \vdash t :: \tau$ does* not *hold, but $\Gamma^*; \Sigma \vdash t :: \tau$* does.

An important completeness claim (we have not formally proved, and indeed have no clear idea of how a proof would go), based on the strategy of injecting harmful $\bot$ whenever possible, is that if **TermFind** finds *no* term, then the naive free theorem in question, i.e., the one omitting all strictness conditions corresponding to non-* type variables in Γ, actually holds.

6 Producing Full Counterexamples

In the introduction we proclaimed the construction of complete counterexamples to naive free theorems. Thus, ideally, if $\alpha, \beta^*, \ldots; \Sigma \Vdash t :: \tau$, then we would want this t to really be a counterexample to the statement given by Theorem 2 for $\alpha, \beta^*, \ldots; \Sigma \vdash t :: \tau$ in **PolyFix***. That is, we would want to establish the negation of: "$(\llbracket t \rrbracket_{\sigma_1}, \llbracket t \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}$ for every θ_1, θ_2, ρ, σ_1, and σ_2 such that $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$, $\rho(\beta) \in Rel^\bot(\theta_1(\beta), \theta_2(\beta))$, ..., and for every $x :: \tau'$ occurring in Σ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$."

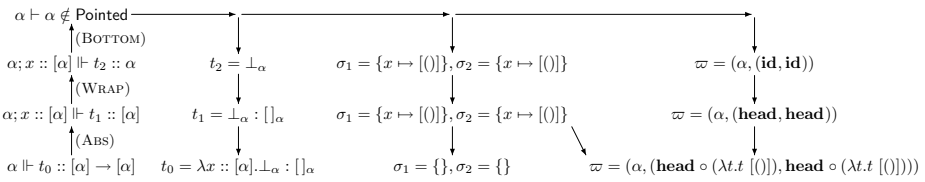
Clearly, Theorem 3 alone does not suffice to do so. Instead, establishing the intended negation requires providing specific environments θ_1, θ_2, ρ, σ_1, and σ_2 that do fulfill all preconditions mentioned above, but not $(\llbracket t \rrbracket_{\sigma_1}, \llbracket t \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}$. One thing we know for sure is that $\rho(\alpha)$ should be nonstrict, i.e., in $Rel \setminus Rel^\bot$, because for every $\rho(\alpha) \in Rel^\bot(\theta_1(\alpha), \theta_2(\alpha))$ the above-quoted statement (and not its negation) would be true by Theorem 1. Regarding the type environments, it suffices for our purposes to let θ_1 and θ_2 map each type variable to the simplest type interpretation that admits both strict and nonstrict functions, namely to the interpretation of the unit type, $\{\bot, ()\}$. This predetermines $\rho(\alpha) \in Rel \setminus Rel^\bot$ to the value of $\lambda x :: ().()$, while we choose identity functions for $\rho(\beta) \in Rel^\bot$.

What remains to be done is to specify σ_1 and σ_2. Here more complex requirements must be met. For example, for every instance of the rule (Bottom) in **TermFind** we need to provide σ_1 and σ_2 such that for every $x :: \tau'$ occurring in Σ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$, but that not $(\llbracket \bot_\tau \rrbracket_{\sigma_1}, \llbracket \bot_\tau \rrbracket_{\sigma_2}) \in \Delta_{\tau,\rho}$. The latter,

$(\bot, \bot) \notin \Delta_{\tau,\rho}$, will be guaranteed by $\Gamma \vdash \tau \notin \mathsf{Pointed}$ and our choices for ρ. But for the former we need to be able to produce for every type τ' concrete values related by $\Delta_{\tau',\rho}$ with our fixed ρ. This production can be achieved based on the structure of τ', but we omit details here. (They are contained in [13].)

For the remaining rules in **TermFind** we mainly need to either propagate already found values unchanged, or manipulate and/or combine them appropriately. For usefully handling the rule (Abs) the introduction of an additional mechanism is required, explained as follows. For this rule we can assume that we have already found some σ_1 and σ_2 with, in particular, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau_1,\rho}$ and $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \notin \Delta_{\tau_2,\rho}$. What we need is to provide σ_1' and σ_2' with $([\![t']\!]_{\sigma_1'}, [\![t']\!]_{\sigma_2'}) \notin \Delta_{\tau_1\to\tau_2,\rho}$ for $t' = (\lambda x :: \tau_1.t)$. This task can be solved by choosing σ_1' and σ_2' to simply be σ_1 and σ_2 without the bindings for x.[4] But then the assignments of values to term variables from the context would not suffice to gather and keep all the information required to establish, and eventually demonstrate to the user of our system, that the overall found term is a counterexample. Hence, we introduce an additional construct, called disrelator, which keeps track of how to manifest, based on the current term, a conflict to the parametricity theorem that was provoked somewhere above in the derivation tree. In the case of the rule (Abs) we precisely record that, in order to "navigate" towards the conflict, the term in the conclusion needs to be applied to the values produced for the additional context term variable in the premise.

We do not go into further details as presented in [13] here, about the treatment of other rules, both due to a lack of space and because these are not needed for the full example run shown in the following figure:



The first two columns illustrate the term search process by applying rules and assembling the result term. The third column shows the environments σ_1 and σ_2 at each stage, and the fourth one shows the disrelators as compositions of single "navigation steps" as obtained from each rule. Additionally, the disrelators remember the specific (subpart of the original) type at which the conflict was provoked, i.e., to which the navigation leads. Consequently, in each row of the figure we see the description of a complete counterexample to the naivified parametricity theorem. In particular, the last row tells us that for the term $\lambda x :: [\alpha].\bot_\alpha : [\,]_\alpha$ of type $[\alpha] \to [\alpha]$ evaluation (in empty environments) leads to values v_1 and v_2 which are unrelated because **head** $(v_1\ [()])$ and **head** $(v_2\ [()])$ are not related by $\rho(\alpha)$. Indeed, both **head** $(v_1\ [()])$ and **head** $(v_2\ [()])$ will be $\bot$, and $(\bot, \bot)$ is not in the relation graph of the function $\lambda x :: ().()$ which we

[4] Then, $[\![t]\!]_{\sigma_1} = ([\![t']\!]_{\sigma_1'}\ \sigma_1(x))$ and $[\![t]\!]_{\sigma_2} = ([\![t']\!]_{\sigma_2'}\ \sigma_2(x))$, and thus $([\![t']\!]_{\sigma_1'}, [\![t']\!]_{\sigma_2'}) \in \Delta_{\tau_1\to\tau_2,\rho}$ would contradict $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau_1,\rho}$ and $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \notin \Delta_{\tau_2,\rho}$.

chose for $\rho(\alpha)$. The result is a counterexample to the naive free theorem (2) from the introduction in very much the same spirit as the counterexample (3) discussed there, but now obtained automatically.

Things can become more complicated and terms found by **TermFind** are not always suitable for full counterexample generation. The reason is the rule (Arrow$\to^\circ$). It splits the derivation tree into two term search branches, and the same term variables may be used in the two branches in different ways. As a consequence, the (σ_1, σ_2) pairs obtained for these two branches separately may disagree on some term variables. To deal with these situations, we keep track of a creation history for values in the σ_i, recording which choices were essential. Whenever the recorded information prevents a merging at the rule (Arrow$\to^\circ$) we abort. Another problem arises from the double use of f in the conclusion of (Arrow$\to^\circ$). Here f might (and sometimes does) fulfill different roles and it is not always possible to provide a single pair of values (to be assigned to f in σ_1 and σ_2) that meets all requirements. Our solution is to switch to a simplified version of (Arrow$\to^\circ$) that omits g and thus the double use of f:

$$\frac{\Gamma \vdash \tau_2, \tau_3 \in \mathsf{Pointed} \qquad \Gamma; \Sigma, x :: \tau_1 \Vdash t_1 :: \tau_2 \qquad \Gamma; \Sigma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \to \tau_2) \to \tau_3 \Vdash [y \mapsto f\ (\lambda x :: \tau_1.t_1)]t_2 :: \tau}$$

The algorithm with this changed rule, history tracking, and value and disrelator construction is what we implemented. It lacks the completeness claim of **TermFind**, but in return we proved correctness in the sense that the counterexamples it produces really contradict the naive free theorems in question. That proof, as well as the proofs of other mentioned results, can be found in [13].

7 Related and Future Work

We have discussed the relation of our approach and results here to [1] and [10] in the introduction and at the end of Section 3. Another related work is [14], where we perform the parametricity-related programme of [10] for the Haskell primitive *seq*, taking the lessons of [7] into account. This provides the basis for a future extension of the work presented here to a more complex language, in particular for producing counterexamples that demonstrate when and why *seq*-imposed side conditions in free theorems are really necessary for a given type. With [14] the part of the development which would be required for that setting up to and including Section 3 of the present paper is already done. Another interesting direction for future work would be to investigate counterexample generation for free theorems in more exotic settings, like the one where nondeterminism and free variables complicate the situation [2].

References

1. L. Augustsson. Putting Curry-Howard to work (Invited talk). At *Approaches and Applications of Inductive Programming*, 2009.

2. J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM Press, 2010.
3. K. Claessen and R.J.M. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming, Proceedings*, pages 268–279. ACM Press, 2000.
4. P. Corbineau. First-order reasoning in the calculus of inductive constructions. In *TYPES 2003, Proceedings*, volume 3085 of *LNCS*, pages 162–177. Springer-Verlag, 2004.
5. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.
6. S. Holdermans and J. Hage. Making "stricterness" more relevant. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 121–130. ACM Press, 2010.
7. P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
8. P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.
9. I. Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976.
10. J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.
11. J.C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Proceedings*, pages 408–423. Springer-Verlag, 1974.
12. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
13. D. Seidel and J. Voigtländer. Automatically generating counterexamples to naive free theorems. Technical Report TUD-FI09-05, Technische Universität Dresden, 2009. `http://www.iai.uni-bonn.de/~jv/TUD-FI09-05.pdf`.
14. D. Seidel and J. Voigtländer. Taming selective strictness. In *Arbeitstagung Programmiersprachen, Proceedings*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, 2009.
15. F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.
16. J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.
17. J. Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.
18. J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
19. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
20. D.A. Wright. A new technique for strictness analysis. In *Theory and Practice of Software Development, Proceedings, Volume 2*, volume 494 of *LNCS*, pages 235–258, 1991.