Parametricity for Haskell with Imprecise Error Semantics[⋆]

Florian Stenger[⋆⋆] and Janis Voigtländer

Technische Universität Dresden
01062 Dresden, Germany
{stenger,voigt}@tcs.inf.tu-dresden.de

Abstract. Error raising, propagation, and handling in Haskell can be imprecise in the sense that a language implementation's choice of local evaluation order, and optimizing transformations to apply, may influence which of a number of potential failure events hidden somewhere in a program is actually triggered. While this has pragmatic advantages from an implementation point of view, it also complicates the meaning of programs and thus requires extra care when reasoning about them. The proper semantic setup is one in which every erroneous value represents a whole set of potential (but not arbitrary) failure causes. The associated propagation rules are somewhat askew to standard notions of program flow and value dependence. As a consequence, standard reasoning techniques are cast into doubt, and rightly so. We study this issue in depth for one such reasoning technique, namely the derivation of free theorems from polymorphic types. We revise and extend the foundational notion of relational parametricity, as well as further material required to make it applicable.

1 Introduction

Functional languages come with a rich set of conceptual tools for reasoning about programs. For example, structural induction and equational reasoning tell us that the standard Haskell functions

$$
\begin{array}{lll}
\mathit{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] & & \\
\mathit{takeWhile}\ p\ [\,] & = [\,] & \\
\mathit{takeWhile}\ p\ (x : y) & \mid\ p\ x & = x : \mathit{takeWhile}\ p\ y \\
 & \mid\ \mathit{otherwise} & = [\,]
\end{array}
$$

and

$$
\begin{array}{l}
\mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\mathit{map}\ h\ [\,] \quad\ = [\,] \\
\mathit{map}\ h\ (x : y) = h\ x : \mathit{map}\ h\ y
\end{array}
$$

[⋆] In *TLCA 2009, Proc.*, volume 5608 of LNCS, pages 294–308. © Springer-Verlag.
[⋆⋆] This author was supported by the DFG under grant VO 1512/1-1.

satisfy the following law for appropriately typed p, h, and l:

$$takeWhile\ p\ (map\ h\ l) = map\ h\ (takeWhile\ (p \circ h)\ l)\,. \tag{1}$$

But programming language reality can be a tough game, leading to unexpected failures of such near-obvious laws. For example, Peyton Jones et al. [5] proposed a design for error handling based on a certain degree of impreciseness. The major implementations GHC and Hugs have integrated this design years ago. However, the resulting semantics breaks law (1). An instantiation showing this is $p = null$, $h = tail$, and $l = [[i] \mid i \leftarrow [1..(div\ 1\ 0)]]$ (or any other immediately failing expression of type list-of-lists), where

$$\begin{array}{ll}
null :: [\alpha] \rightarrow \mathsf{Bool} & tail :: [\alpha] \rightarrow [\alpha] \\
null\ [\,] \quad\;\;\; = \mathsf{True} & tail\ [\,] \quad\;\;\; = error\ \text{“tail: empty list”} \\
null\ (x : y) = \mathsf{False} & tail\ (x : y) = y
\end{array}$$

are standard Haskell functions as well. The problem with (1) now is that its left-hand side yields exactly the “divide by zero”-error coming from l, whereas its right-hand side may also yield the “tail: empty list”-error. This is so due to the semantics of pattern-matching in the mentioned design [5]. In short, it prescribes that when pattern-matching on an erroneous value as scrutinee, not only are any errors associated with it propagated, but, in addition, the branches of the pattern-match are investigated in “error-finding mode” to detect any errors that may arise there independently of the scrutinee. This is done to give the language implementation more freedom in arranging computations, thus allowing more transformations on the code prior to execution. But here it means that when $takeWhile\ (null \circ tail)$ encounters an erroneous value, also $(null \circ tail)\ x$ is evaluated, with x bound to a special value $Bad\ \emptyset$ that exists only to trigger the error-finding mode. And indeed, the application of $tail$ on that x raises the “tail: empty list”-error, which is propagated by $null$ and then unioned with the “divide by zero”-error from l. In contrast, $takeWhile\ null$ on an erroneous value does not add any further errors, because the definition of $null$ raises none. And, on both sides of (1), $map\ h$ only ever propagates, but never introduces errors.

Thus, if we do not want to take the risk of introducing previously non-existent errors, we cannot use (1) as a transformation from left to right, even though this might have been beneficial (by bringing p and h together for further analysis or for subsequent transformations potentially improving efficiency). The supposed semantic equivalence simply does not hold. So impreciseness in the semantics has its price, and if we are not ready to abandon the overall design (which would be tantamount to taking away considerable freedom from language implementers), then we must learn how to cope with it when reasoning about programs.

The above discussion regarding a concrete instantiation of p, h, and l gives negative information only, namely that (1) may break down in some cases. It does not provide any positive information about conditions on p, h, and l under which (1) actually *is* a semantic equivalence. Moreover, it is relative to the particular definition of $takeWhile$ given at the very beginning, whereas laws like (1) are often derived more generally as *free theorems* [7,9] from types alone, without considering concrete definitions. In this paper, we develop the theory of free theorems for Haskell with imprecise error semantics. This continues earlier work [1] for Haskell with all potential error causes (including non-termination) conflated into a single erroneous value $\perp$. That earlier work indicates that, in this setting, (1) is a semantic equivalence provided $p \neq \perp$ and h is strict and total in the sense that $h \perp = \perp$ and for every $x \neq \perp$, $h\ x \neq \perp$. The task before us involves finding the right generalizations of such conditions for a setting in which not all errors are equal. Questions like the following ones arise:

- From which erroneous values should p be different?
- For strictness, is it enough that h preserves the least element $\perp$, which in the design of Peyton Jones et al. [5] denotes the union of all error causes, including non-termination?
- Or do we need that also every other erroneous value (denoting a collection of only some potential error causes, maybe just a singleton set) is mapped to an erroneous one? To the same one? Or to $\perp$?
- For totality, is it enough that non-$\perp$ values are mapped to non-$\perp$ values, including possibly to non-$\perp$ but still erroneous values?
- Or do we need that h maps non-erroneous values only to non-erroneous ones?

We should not expect trivial answers to these questions. The two settings are simply too different. In particular, it is worth pointing out that the failure of (1) occurs for a very innocently-looking definition of *takeWhile* here. Note that *takeWhile* as defined does not, by itself, introduce any errors or non-termination, nor does it use selective strictness via Haskell's *seq*-primitive. Thus, the features that made life hard before are actually absent, and still the law breaks down.[1] In fact, were it not for the imprecise error semantics, (1) would hold for the given definition of *takeWhile* as a semantic equivalence for arbitrary p, h, and l, even ones involving $\perp$ and *seq* in arbitrary ways and without strictness or totality conditions. By contraposition, this indicates that genuinely new challenges are posed by the imprecise error semantics.

Fortunately, we are not left groping in the dark. Our investigation can be very much goal-directed by studying proof cases of the *(relational) parametricity theorem* [7,9,6], which is the foundation for all free theorems, and trying to adapt the proof to the imprecise error setting. This leads us to discover, among other formal details and ingredients, the appropriate generalized conditions sought above (first as restrictions on relations, then specialized to the function level).

Note that even though we do not deal with exception handling in the functions *for which* we derive free theorems, our results are nevertheless immediately relevant as well in the larger context of programs that *do* error recovery (in the IO monad; see the description by Peyton Jones [4, Section 5.2]). Just imagine alternately the left- or right-hand side of the offending instantiation of (1) in the

[1] Indeed, Johann and Voigtländer [1] had to add rather ad-hoc occurrences of *seq* to a definition of the *filter*-function (of same type as *takeWhile*) to "provoke" failures of the corresponding standard free theorem. Here, instead, even the most natural specification of *takeWhile* leads to problems.

place of the "$\cdots$" in the following code snippet:

$$\begin{array}{l}\mathsf{Control.Exception}.\mathit{catch}\ (\mathit{evaluate}\ \cdots)\ (\lambda s \to \mathit{if}\ \ s \neq \mathsf{ErrorCall}\ \text{"tail: empty list"} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathit{return}\ [[42]] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \mathit{return}\ [\,])\end{array}$$

Then depending on whether the left- or right-hand side of (1) is put there, we might observe different non-erroneous program outcomes. This is even more severe than "just" a confusion between different erroneous values.

With the results from this paper both kinds of problems are settled. For example, we will derive (cf. Example 1) that (1) is a true semantic equivalence provided p and h are non-erroneous, h acts as identity on erroneous values, and h never maps a non-erroneous value to an erroneous one. Similar fixes can be obtained for other free theorems. The accompanying technical report [8] goes on to establish "inequational" parametricity theorems, including one for the refinement order of Moran et al. [3]. Then, for example, slightly weaker conditions than those mentioned above suffice for a variant of (1) in which the left-hand side is only stated to semantically approximate the right-hand side. The technical report also makes some initial steps into the realm of exceptions as first class citizens by integrating a primitive (Haskell's *mapException*) that allows manipulating already raised errors (respectively, their descriptive arguments) from inside the language.

The work most closely related to that reported here is the recent one of Johann and Voigtländer [2], which also studies relational parametricity for a setting in which different failure causes are semantically distinguished. However, that earlier work does not consider the imprecise error semantics embodied in the mentioned Haskell implementations. Rather, error treatment there is completely deterministic, but results are given modulo a presumed, and then fixed, order on erroneous values. It might be tempting to try to encode the "contents" of erroneous values in the imprecise error semantics, namely sets of error causes, into the unstructured erroneous values of the deterministic setup. Then one could try to choose the order on these unstructured values to agree with the reversed subset order prescribed by Peyton Jones et al. [5] on the encoded sets. But this approach cannot faithfully model how errors are propagated and combined in the imprecise error semantics, e.g., by taking unions in the semantics of pattern-matching. In fact, (1) is a semantic equivalence in the setting of Johann and Voigtländer [2] (for the given definition of *takeWhile*), no matter what order on erroneous values is chosen. This means that their formal development is fundamentally unsuited to make the semantic distinctions that need to be made here.

The remainder of the paper is structured as follows. Section 2 introduces a Haskell-like calculus with imprecise error semantics. Section 3.1 recalls the standard approach to relational parametricity. Sections 3.2 and 3.3 adapt it to the imprecise error setting, and Section 3.4 shows how to derive revised free theorems. Section 4 concludes with an outlook on future work.

2 Imprecise Error Semantics

We consider a polymorphic lambda-calculus that corresponds to Haskell with a semantics that distinguishes between different causes of failure. The syntax of types and terms is given as follows, where α ranges over type variables, x over term variables, and n over the integers:

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

Note that the calculus is explicitly typed and that type abstraction and application are explicit in the syntax as well. General recursion is captured via a fixpoint combinator, while selective strictness (à la Haskell's *seq*) is provided via a strict-let construct. That construct's standard semantics is to evaluate the term bound to the term variable, independently of its use in the body term, and to eventually return the evaluation of the latter, potentially reusing the evaluation of the former term.

Fig. 1 gives the typing rules for the calculus.[2] Standard conventions apply. 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. The explicit type information in the syntax of empty lists ensures that for every Γ and t there is at most one τ with $\Gamma \vdash t : \tau$.

$$\Gamma, x : \tau \vdash x : \tau \qquad \Gamma \vdash n : \mathsf{Int} \qquad \Gamma \vdash [\,]_\tau : [\tau] \qquad \Gamma \vdash \mathbf{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Int} \qquad \Gamma \vdash t_2 : \mathsf{Int}}{\Gamma \vdash (t_1 + t_2) : \mathsf{Int}} \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : [\tau]}{\Gamma \vdash (t_1 : t_2) : [\tau]} \qquad \frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau}$$

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

$$\frac{\Gamma \vdash t : \forall\alpha.\tau_1}{\Gamma \vdash (t\ \tau_2) : \tau_1[\tau_2/\alpha]} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2} \qquad \Gamma \vdash \mathbf{error} : \forall\alpha.\mathsf{Int} \to \alpha$$

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

Fig. 1. Typing Rules.

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

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

[2] Note that, to simplify the presentation, we deviate from Haskell by using integers rather than strings as descriptive arguments for **error**.

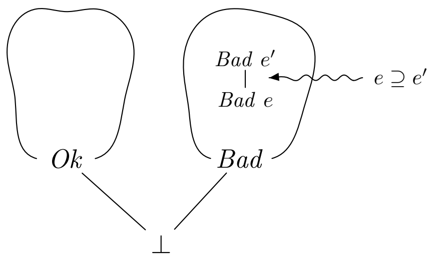
We use a denotational semantics that extends the one given by Peyton Jones et al. [5], our main extension being that we formalize the treatment of polymorphic types. Peyton Jones et al.'s main innovation, and the reason for calling the semantics "imprecise", is the use of *sets* of possible failure causes. Formally, let

$$\mathcal{E} = \{\mathsf{ErrorCall}\ n \mid n \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}\}$$

and $\mathcal{E}^{nt} = \{\mathsf{NonTermination}\} \cup \mathcal{E}$, where NonTermination and ErrorCall are descriptive tags for use in the denotational semantics but without direct syntactical counterparts in the underlying calculus. The set of all erroneous values is then $V_{err} = \{Bad\ e \mid e \in \mathcal{P}(\mathcal{E}) \cup \{\mathcal{E}^{nt}\}\}$[3] and its elements are ordered by

$$Bad\ e \sqsubseteq Bad\ e' \ \text{ iff } \ e \supseteq e' \,. \tag{2}$$

The operation *lift* maps complete partial orders to so-called error-lifted cpos (henceforth, for short, *elcpos*): $lift\ S = V_{err} \cup \{Ok\ s \mid s \in S\}$. The approximation order on such an elcpo is given by (2) on erroneous values, by taking over the order from S for non-erroneous values, and by mandating that $\bot = Bad\ \mathcal{E}^{nt}$ is below all, even non-erroneous, values, while otherwise erroneous and non-erroneous values are pairwise incomparable. Illustrated as a diagram, the structure of an elcpo is as follows:



With the above definitions in place, types are interpreted as elcpos as follows, where θ is a mapping from type variables to elcpos:

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha)\\
[\![\mathsf{Int}]\!]_\theta &= lift\,\{\ldots, -2, -1, 0, 1, 2, \ldots\}\\
[\![[\tau]]\!]_\theta &= gfp\,(\lambda S. lift\,(\{[\,]\} \cup \{a : b \mid a \in [\![\tau]\!]_\theta,\, b \in S\}))\\
[\![\tau_1 \to \tau_2]\!]_\theta &= lift\,\{f : [\![\tau_1]\!]_\theta \to [\![\tau_2]\!]_\theta\}\\
[\![\forall\alpha.\tau]\!]_\theta &= lift\,\{g \mid \forall D \text{ elcpo. } (g\ D) \in [\![\tau]\!]_{\theta[\alpha\mapsto D]} \setminus V_{err}\}\,.
\end{aligned}$$

The first four lines are consistent with a standard semantics featuring only a single erroneous value $\bot$ at every type. The complete partial order lifted in the definition of $[\![\mathsf{Int}]\!]_\theta$ is the flat one without ordering between integers. For list types, prior to lifting, $[\,]$ is only related to itself, while the ordering between

[3] Note that if the e in a $(Bad\ e) \in V_{err}$ contains NonTermination, then it must also contain every other possible failure cause.

"$- : -$"-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. The elements in the set lifted in the definition of $[\![\forall\alpha.\tau]\!]_\theta$ are again ordered point-wise (i.e., $g_1 \sqsubseteq g_2$ iff for every elcpo D, $g_1\ D \sqsubseteq g_2\ D$). Note that, in this last line, by subtracting V_{err} from the possible ranges of g, we mandate that a non-erroneous polymorphic value does not have any erroneous instantiation. In particular, we thus exclude, as in Haskell, polymorphic values of which the instantiation at some type is erroneous and at some other type is non-erroneous. More specifically, an erroneous polymorphic value exhibits exactly the same potential failing behavior in each of its instantiations. Of course, ensuring all this also depends on the term semantics, to be considered next:[4]

$$\begin{array}{ll}
[\![x]\!]_{\theta,\sigma} & = \sigma(x) \\
[\![n]\!]_{\theta,\sigma} & = Ok\ n \\
[\![t_1 + t_2]\!]_{\theta,\sigma} & = \\
\multicolumn{2}{l}{\quad \begin{cases} Ok\ (n_1 + n_2) & \text{if } [\![t_1]\!]_{\theta,\sigma} = Ok\ n_1,\ [\![t_2]\!]_{\theta,\sigma} = Ok\ n_2 \\ Bad\ (E([\![t_1]\!]_{\theta,\sigma}) \cup E([\![t_2]\!]_{\theta,\sigma})) & \text{otherwise} \end{cases}} \\
[\![[\,]_\tau]\!]_{\theta,\sigma} & = Ok\ [\,] \\
[\![t_1 : t_2]\!]_{\theta,\sigma} & = Ok\ ([\![t_1]\!]_{\theta,\sigma} : [\![t_2]\!]_{\theta,\sigma}) \\
\multicolumn{2}{l}{[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\ ;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma} =} \\
\multicolumn{2}{l}{\quad \begin{cases} [\![t_1]\!]_{\theta,\sigma} & \text{if } [\![t]\!]_{\theta,\sigma} = Ok\ [\,] \\ [\![t_2]\!]_{\theta,\sigma[x_1 \mapsto a,\, x_2 \mapsto b]} & \text{if } [\![t]\!]_{\theta,\sigma} = Ok\ (a : b) \\ Bad\ (e \cup E([\![t_1]\!]_{\theta,\sigma}) \cup E([\![t_2]\!]_{\theta,\sigma[x_1 \mapsto Bad\,\emptyset,\, x_2 \mapsto Bad\,\emptyset]})) & \text{if } [\![t]\!]_{\theta,\sigma} = Bad\ e \end{cases}} \\
[\![\lambda x : \tau.t]\!]_{\theta,\sigma} & = Ok\ (\lambda a.[\![t]\!]_{\theta,\sigma[x \mapsto a]}) \\
[\![t_1\ t_2]\!]_{\theta,\sigma} & = [\![t_1]\!]_{\theta,\sigma}\ \$\ [\![t_2]\!]_{\theta,\sigma} \\
[\![\Lambda\alpha.t]\!]_{\theta,\sigma} & = \begin{cases} Ok\ (\lambda D.[\![t]\!]_{\theta[\alpha \mapsto D],\sigma}) & \text{if } [\![t]\!]_{\theta[\alpha \mapsto V_{err}],\sigma} = Ok\ v \\ Bad\ e & \text{if } [\![t]\!]_{\theta[\alpha \mapsto V_{err}],\sigma} = Bad\ e \end{cases} \\
[\![t\ \tau]\!]_{\theta,\sigma} & = [\![t]\!]_{\theta,\sigma}\ \$\$\ [\![\tau]\!]_\theta \\
[\![\mathbf{fix}]\!]_{\theta,\sigma} & = Ok\ (\lambda D.Ok\ (\lambda h.\bigsqcup((h\,\$)^i\ \bot))) \\
[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]_{\theta,\sigma} & = \begin{cases} [\![t_2]\!]_{\theta,\sigma[x \mapsto Ok\,v]} & \text{if } [\![t_1]\!]_{\theta,\sigma} = Ok\ v \\ Bad\ (e \cup E([\![t_2]\!]_{\theta,\sigma[x \mapsto Bad\,\emptyset]})) & \text{if } [\![t_1]\!]_{\theta,\sigma} = Bad\ e \end{cases} \\
[\![\mathbf{error}]\!]_{\theta,\sigma} & = Ok\ (\lambda D.Ok\ (\lambda a. \begin{cases} Bad\ \{\mathsf{ErrorCall}\ n\} & \text{if } a = Ok\ n \\ Bad\ e & \text{if } a = Bad\ e \end{cases}))
\end{array}$$

Most of the above definitions are straightforward. They use λ for denoting anonymous functions, and the following two operators:

$$h\ \$\ a = \begin{cases} f\ a & \text{if } h = Ok\ f \\ Bad\ (e \cup E(a)) & \text{if } h = Bad\ e \end{cases}, \quad \text{where} \quad E(a) = \begin{cases} \emptyset & \text{if } a = Ok\ v \\ e & \text{if } a = Bad\ e\,, \end{cases}$$

[4] Here σ is a mapping from term variables to values.

and

$$h \mathbin{\$\$} D = \begin{cases} g\ D & \text{if } h = \mathit{Ok}\ g \\ \mathit{Bad}\ e & \text{if } h = \mathit{Bad}\ e\,. \end{cases}$$

One crucial point here, taken from Peyton Jones et al. [5], is that application of an erroneous function value incurs all potential failures of the argument as well. We also essentially use their definitions of $[\![t_1 + t_2]\!]_{\theta,\sigma}$ and $[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma}$, except that we do not check for overflow in the case of addition. To bring about erroneous values other than $\bot$ in the first place, we have the obvious definition of $[\![\mathbf{error}]\!]_{\theta,\sigma}$. The expression $\bigsqcup((h\,\$)^i\ \bot)$ in the definition for **fix** means the supremum of the chain $\bot \sqsubseteq h\ \$\ \bot \sqsubseteq h\ \$\,(h\ \$\ \bot) \cdots$. For $[\![\Lambda\alpha.t]\!]_{\theta,\sigma}$, we first need to analyze the semantics of t to sort out the exceptional case that every D is mapped to an (actually, one and the same) erroneous value, in which case the semantics of $\Lambda\alpha.t$ should itself be that erroneous value, as explicitly added via *lift* in the definition of $[\![\forall\alpha.\tau]\!]_\theta$. Due to the observed uniqueness, it is not actually necessary to check the behavior for every D. Instead, the test can be performed with a single, arbitrary elcpo. We choose the simplest one, namely just V_{err}. If we find that we are not in the exceptional case, the denotation is just the standard one, but appropriately tagged via *Ok*. Finally, the definition of $[\![\mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2]\!]_{\theta,\sigma}$ follows the standard one, but similarly to the definition of $[\![\mathbf{case}\ t\ \mathbf{of}\ \{[\,] \to t_1\,;\ x_1 : x_2 \to t_2\}]\!]_{\theta,\sigma}$, and in line with the operational semantics of Moran et al. [3], t_2 is evaluated in "error-finding mode" to contribute further potential failure causes in case t_1 is already erroneous. Altogether, we have that if $\Gamma \vdash t : \tau$ and $\sigma(x) \in [\![\tau']\!]_\theta$ for every $x : \tau'$ occurring in Γ, then $[\![t]\!]_{\theta,\sigma} \in [\![\tau]\!]_\theta$.

3 Parametricity

3.1 The Standard Logical Relation

The key to parametricity results is the definition of a family of relations by induction on a calculus' type structure. If we were to abandon the primitive **error**, and thus return to a setting without distinguishing error causes (i.e., with only one erroneous value $\bot$), then the appropriate such *logical relation* would be as follows, where ρ is a mapping from type variables to binary relations between pointed complete partial orders:

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

We use *id* to denote identity relations. The operation *list* takes a relation $\mathcal{R}$ and maps it to

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

where again the greatest fixpoint is taken. $\mathit{Rel}(D_1, D_2)$ collects all relations between D_1 and D_2 that are *strict*, *continuous*, and *bottom-reflecting*. Strictness and continuity are just the standard notions (i.e., membership of the pair $(\bot, \bot)$ and closure under suprema). A relation $\mathcal{R}$ is bottom-reflecting if $(a, b) \in \mathcal{R}$ implies that $a = \bot$ iff $b = \bot$. The corresponding explicit condition on f and g in the definition of $\Delta_{\tau_1 \to \tau_2, \rho}$ serves the purpose of ensuring that bottom-reflection is preserved throughout the logical relation.

Overall, for that $\bot$-only setting, reasoning like Johann and Voigtländer [1] do gives the following important lemma (by induction on τ), where Rel is the union of all $\mathit{Rel}(D_1, D_2)$. That lemma is crucial for then proving the parametricity theorem.

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

Theorem 1. *If $\Gamma \vdash t : \tau$, then for every θ_1, θ_2, ρ, σ_1, and σ_2 such that*

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

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

For reference, the proof of Theorem 1 for the setting without **error**, spelling out more formally the "narrative" of Johann and Voigtländer [1], is given in Appendix A of the accompanying technical report [8].

3.2 Towards Appropriate Restrictions on Relations

To establish an analogue of Theorem 1 for the setting including non-$\bot$ errors, and their deliberately imprecise semantics, we first need to determine just the right set of restrictions to impose on relational interpretations of types. Above, we required strict, continuous, and bottom-reflecting relations. It seems reasonable that continuity will still be required as we still have general recursion via the fixpoint combinator. But for strictness and bottom-reflection, the situation is less clear when we have more than a single erroneous value $\bot$ to consider.

For example, strictness currently only states that the pair $(\bot, \bot)$ (i.e., the pair $(\mathit{Bad}\ \mathcal{E}^{nt}, \mathit{Bad}\ \mathcal{E}^{nt})$) should be contained in every relation. But what about other erroneous values? Should any pair of them be related? Or only identical ones? Or is inclusion of $(\bot, \bot)$ actually enough?

The best way to answer such questions is to go through the proof of Theorem 1 and see where changes in the calculus and its semantics might require a change in the proof. In our case, it of course makes most sense to study the impact of the new primitive **error** first. Recalling its typing rule, we will have to prove that, for every θ_1, θ_2, ρ, σ_1, and σ_2 such that

- for every α occurring in Γ, $\rho(\alpha)$ is an appropriately restricted relation between $\theta_1(\alpha)$ and $\theta_2(\alpha)$, and
- for every $x : \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau',\rho}$,

we have $(\llbracket \mathbf{error} \rrbracket_{\theta_1,\sigma_1}, \llbracket \mathbf{error} \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\forall\alpha.\mathsf{Int}\to\alpha,\rho}$.

By the definition of Δ, this will require to establish that for every D_1, D_2, and (appropriate) $\mathcal{R}$, $(\llbracket \mathbf{error} \rrbracket_{\theta_1,\sigma_1}\ \$\$\ D_1, \llbracket \mathbf{error} \rrbracket_{\theta_2,\sigma_2}\ \$\$\ D_2) \in \Delta_{\mathsf{Int}\to\alpha,\rho[\alpha\mapsto\mathcal{R}]}$. Further unfolding the current definition of Δ tells us that we will have to show that

$$\llbracket \mathbf{error} \rrbracket_{\theta_1,\sigma_1}\ \$\$\ D_1 = \bot \ \text{ iff } \ \llbracket \mathbf{error} \rrbracket_{\theta_2,\sigma_2}\ \$\$\ D_2 = \bot \tag{3}$$

(or a similar statement involving also non-$\bot$ erroneous values) and that for every $(a,b) \in \Delta_{\mathsf{Int},\rho[\alpha\mapsto\mathcal{R}]}$, $(\llbracket \mathbf{error} \rrbracket_{\theta_1,\sigma_1}\ \$\$\ D_1\ \$\ a, \llbracket \mathbf{error} \rrbracket_{\theta_2,\sigma_2}\ \$\$\ D_2\ \$\ b) \in \mathcal{R}$. Taking into account that the integer type should still be interpreted by an identity relation, and using the semantics definitions given in Section 2, the latter is the same as requiring that for every $a \in \mathit{lift}\,\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the value

$$\begin{cases} \mathit{Bad}\ \{\mathsf{ErrorCall}\ n\} & \text{if } a = \mathit{Ok}\ n \\ \mathit{Bad}\ e & \text{if } a = \mathit{Bad}\ e \end{cases}$$

is related to itself by $\mathcal{R}$, which is equivalent to requiring that every erroneous value is related to itself by $\mathcal{R}$. Therefore, we propose to generalize the notion of strictness as follows, with $\mathit{id}_{V_{err}} = \{(a,a) \mid a \in V_{err}\}$.

Definition 1. *A relation $\mathcal{R}$ is* error-strict *if $\mathit{id}_{V_{err}} \subseteq \mathcal{R}$.*

Similar questions as for strictness arise for bottom-reflection in the presence of different failure causes. Is it enough to maintain that two related values are either both $\bot$ or else neither one of them is? Or should we generalize by requiring that either both are erroneous or else neither one of them is? Or should we be even more demanding by even expecting that only equal failure causes (or sets thereof) are related?

The relevant proof case to check here is the one for the strict-let construct, because selective strictness was what necessitated bottom-reflection in the first place [1]. Recall the typing rule. Inside the proof of an analogue of Theorem 1 by induction over typing derivations we will have to establish, for the induction conclusion in this case, that, for θ_1, θ_2, ρ, σ_1, and σ_2 as above, we have $(\llbracket \mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2 \rrbracket_{\theta_1,\sigma_1}, \llbracket \mathbf{let!}\ x = t_1\ \mathbf{in}\ t_2 \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau_2,\rho}$. The semantics from Section 2 tells us that the two values in the relation of which we are interested here are equal to

$$\begin{cases} \llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x\mapsto \mathit{Ok}\, v_1]} & \text{if } \llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = \mathit{Ok}\ v_1 \\ \mathit{Bad}\ (e_1 \cup E(\llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x\mapsto \mathit{Bad}\,\emptyset]})) & \text{if } \llbracket t_1 \rrbracket_{\theta_1,\sigma_1} = \mathit{Bad}\ e_1 \end{cases}$$

and

$$\begin{cases} \llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x\mapsto \mathit{Ok}\, v_2]} & \text{if } \llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = \mathit{Ok}\ v_2 \\ \mathit{Bad}\ (e_2 \cup E(\llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x\mapsto \mathit{Bad}\,\emptyset]})) & \text{if } \llbracket t_1 \rrbracket_{\theta_2,\sigma_2} = \mathit{Bad}\ e_2\,, \end{cases}$$

respectively. The role of bottom-reflection in the $\bot$-only setting is to ensure, via the induction hypothesis corresponding to the precondition $\Gamma \vdash t_1 : \tau_1$, viz.

$$(\llbracket t_1 \rrbracket_{\theta_1,\sigma_1}, \llbracket t_1 \rrbracket_{\theta_2,\sigma_2}) \in \Delta_{\tau_1,\rho}\,, \tag{4}$$

that the same branch is chosen in (the analogues of) the two case distinctions above. Here the same can be achieved by introducing an auxiliary function extracting the tag of a value as follows:

$$T(a) = \begin{cases} \mathit{Ok} & \text{if } a = \mathit{Ok}\ v \\ \mathit{Bad} & \text{if } a = \mathit{Bad}\ e\,, \end{cases}$$

and generalizing bottom-reflection in such a way that related values are always required to have the same image under T.

This suffices in the case that $[\![t_1]\!]_{\theta_1,\sigma_1} = \mathit{Ok}\ v_1$ and $[\![t_1]\!]_{\theta_2,\sigma_2} = \mathit{Ok}\ v_2$, because we then get the desired $([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto \mathit{Ok}\,v_1]}, [\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto \mathit{Ok}\,v_2]}) \in \Delta_{\tau_2,\rho}$ from $(\mathit{Ok}\ v_1, \mathit{Ok}\ v_2) \in \Delta_{\tau_1,\rho}$ (cf. (4)) and the induction hypothesis corresponding to the precondition $\Gamma, x : \tau_1 \vdash t_2 : \tau_2$, namely that for every $(b, c) \in \Delta_{\tau_1,\rho}$,

$$([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto b]}, [\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto c]}) \in \Delta_{\tau_2,\rho}\,. \tag{5}$$

However, in the case that $[\![t_1]\!]_{\theta_1,\sigma_1} = \mathit{Bad}\ e_1$ and $[\![t_1]\!]_{\theta_2,\sigma_2} = \mathit{Bad}\ e_2$, we need to show that

$$(\mathit{Bad}\ (e_1 \cup E([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto \mathit{Bad}\,\emptyset]})), \mathit{Bad}\ (e_2 \cup E([\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto \mathit{Bad}\,\emptyset]}))) \in \Delta_{\tau_2,\rho}\,, \tag{6}$$

and do not yet have the means for doing so. Note that a supposed error-strictness of $\Delta_{\tau_2,\rho}$ would only allow us to conclude the desired membership if the sets $e_1 \cup E([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto \mathit{Bad}\,\emptyset]})$ and $e_2 \cup E([\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto \mathit{Bad}\,\emptyset]})$ were equal. Revising the notion of error-strictness to guarantee that indeed any two erroneous values are related, independently of the sets of possible failures they represent, would risk completely blurring any distinction between different failure causes, and thus is not an acceptable option. Instead, the proposed generalization of bottom-reflection is strengthened in a very natural way. Rather than just requiring that two related values always have the same image under T, we expect the same under E.

Definition 2. *A relation $\mathcal{R}$ is* error-reflecting *if $(a, b) \in \mathcal{R}$ implies that $T(a) = T(b)$ and $E(a) = E(b)$.*[5]

Then, (4) and the assumption that $\Delta_{\tau_1,\rho}$ is error-reflecting imply that in the case $[\![t_1]\!]_{\theta_1,\sigma_1} = \mathit{Bad}\ e_1$ and $[\![t_1]\!]_{\theta_2,\sigma_2} = \mathit{Bad}\ e_2$ we even have $e_1 = e_2$. Moreover, (5) and $(\mathit{Bad}\ \emptyset, \mathit{Bad}\ \emptyset) \in \Delta_{\tau_1,\rho}$ (cf. supposed error-strictness of $\Delta_{\tau_1,\rho}$) give $([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto \mathit{Bad}\,\emptyset]}, [\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto \mathit{Bad}\,\emptyset]}) \in \Delta_{\tau_2,\rho}$, and thus by supposed error-reflection of $\Delta_{\tau_2,\rho}$, $E([\![t_2]\!]_{\theta_1,\sigma_1[x\mapsto \mathit{Bad}\,\emptyset]}) = E([\![t_2]\!]_{\theta_2,\sigma_2[x\mapsto \mathit{Bad}\,\emptyset]})$ as well, which finally establishes (6), without having to revise the notion of error-strictness.

3.3 The New Logical Relation and its Parametricity Theorem

We have been led to focus on relations that are error-strict, continuous, and error-reflecting. Clearly, ensuring that these restrictions are preserved will require

[5] Note that $E(a) = E(b)$ does not imply $T(a) = T(b)$, as can be seen by taking $a = \mathit{Bad}\ \emptyset$ and $b = \mathit{Ok}\ v$.

changes to the definition of Δ. For example, the operation *list* used in Section 3.1 does not suffice anymore, but it is easy enough to replace it as follows:

$$\begin{aligned} \mathit{list}^{err}\,\mathcal{R} = \mathit{gfp}\,(\lambda\mathcal{S}.\mathit{id}_{V_{err}} &\cup \{(\mathit{Ok}\ [\,],\mathit{Ok}\ [\,])\} \\ &\cup \{(\mathit{Ok}\ (a:b),\mathit{Ok}\ (c:d)) \mid (a,c)\in\mathcal{R},\ (b,d)\in\mathcal{S}\})\,. \end{aligned}$$

For the case of function types, we clearly need an appropriate replacement for the "$f = \bot$ iff $g = \bot$"-condition occurring in the definition of $\Delta_{\tau_1\to\tau_2,\rho}$. It might seem that, to guarantee error-reflection (instead of bottom-reflection, as earlier), we will have to require "$T(f) = T(g)$ and $E(f) = E(g)$". But actually it turns out that requiring just "$T(f) = T(g)$" is enough, as then the other conjunct can be established from relatedness of $f\ \$\ a$ and $g\ \$\ b$ for related a and b (see below). For the case of polymorphic types, we clearly have to restrict the relations that we quantify over to error-strict, continuous, and error-reflecting ones. To this end, for given elcpos D_1 and D_2, let $\mathit{Rel}^{err}(D_1,D_2)$ collect all relations between them that are error-strict, continuous, and error-reflecting. (Also, let Rel^{err} be the union of all $\mathit{Rel}^{err}(D_1,D_2)$.) Overall, we obtain the new logical relation defined as follows:

$$\begin{aligned} \Delta^{err}_{\alpha,\rho} &= \rho(\alpha) \\ \Delta^{err}_{\mathsf{Int},\rho} &= \mathit{id}_{\mathit{lift}\{\ldots,-2,-1,0,1,2,\ldots\}} \\ \Delta^{err}_{[\tau],\rho} &= \mathit{list}^{err}\,\Delta^{err}_{\tau,\rho} \\ \Delta^{err}_{\tau_1\to\tau_2,\rho} &= \{(f,g) \mid T(f)=T(g),\ \forall (a,b)\in\Delta^{err}_{\tau_1,\rho}.\ (f\ \$\ a, g\ \$\ b)\in\Delta^{err}_{\tau_2,\rho}\} \\ \Delta^{err}_{\forall\alpha.\tau,\rho} &= \{(u,v) \mid \forall D_1,D_2,\mathcal{R}\in\mathit{Rel}^{err}(D_1,D_2). \\ &\qquad\qquad (u\ \$\$\ D_1, v\ \$\$\ D_2)\in\Delta^{err}_{\tau,\rho[\alpha\mapsto\mathcal{R}]}\} \end{aligned}$$

Now we can state the following analogue of Lemma 1.

Lemma 2. *If ρ maps only to relations in Rel^{err}, then $\Delta^{err}_{\tau,\rho}\in\mathit{Rel}^{err}$.*

The proof is mostly routine, but we briefly sketch a few interesting parts related to the treatment of erroneous values:

– Error-strictness of $\Delta^{err}_{\tau_1\to\tau_2,\rho}$ follows from error-reflection of $\Delta^{err}_{\tau_1,\rho}$ and error-strictness of $\Delta^{err}_{\tau_2,\rho}$, because for every $e\in\mathcal{P}(\mathcal{E})\cup\{\mathcal{E}^{nt}\}$ and a,b with $E(a) = E(b)$, we have $((\mathit{Bad}\ e)\ \$\ a, (\mathit{Bad}\ e)\ \$\ b)\in\mathit{id}_{V_{err}}$.
– Error-reflection of $\Delta^{err}_{\tau_1\to\tau_2,\rho}$ follows from error-strictness of $\Delta^{err}_{\tau_1,\rho}$ and error-reflection of $\Delta^{err}_{\tau_2,\rho}$, because for every $e,e'\in\mathcal{P}(\mathcal{E})\cup\{\mathcal{E}^{nt}\}$, we have that $E((\mathit{Bad}\ e)\ \$\,(\mathit{Bad}\ \emptyset)) = E((\mathit{Bad}\ e')\ \$\,(\mathit{Bad}\ \emptyset))$ implies $e = e'$.
– Error-strictness of $\Delta^{err}_{\forall\alpha.\tau,\rho}$ follows from error-strictness of $\Delta^{err}_{\tau,\rho[\alpha\mapsto\mathcal{R}]}$ for every error-strict, continuous, and error-reflecting relation $\mathcal{R}$, because for every $e\in\mathcal{P}(\mathcal{E})\cup\{\mathcal{E}^{nt}\}$ and elcpos D_1 and D_2, $((\mathit{Bad}\ e)\ \$\$\ D_1, (\mathit{Bad}\ e)\ \$\$\ D_2)\in\mathit{id}_{V_{err}}$.
– Error-reflection of $\Delta^{err}_{\forall\alpha.\tau,\rho}$ follows from error-reflection of $\Delta^{err}_{\tau,\rho[\alpha\mapsto\mathcal{R}]}$ for every error-strict, continuous, and error-reflecting relation $\mathcal{R}$, because for every (erroneous or non-erroneous) value h in $[\![\forall\alpha.\tau]\!]_\theta$ for some θ, and every elcpo D, we have $T(h\ \$\$\ D) = T(h)$ and $E(h\ \$\$\ D) = E(h)$.

Being assured of Lemma 2 is nice, but not our ultimate goal. Rather, we want the following analogue of Theorem 1.

Theorem 2. *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, $\rho(\alpha) \in Rel^{err}(\theta_1(\alpha), \theta_2(\alpha))$, and*
- *for every $x : \tau'$ occurring in Γ, $(\sigma_1(x), \sigma_2(x)) \in \Delta^{err}_{\tau',\rho}$,*

we have $(\llbracket t \rrbracket_{\theta_1,\sigma_1}, \llbracket t \rrbracket_{\theta_2,\sigma_2}) \in \Delta^{err}_{\tau,\rho}$.

Of course, Lemma 2 plays an important role in the proof, in particular in the inductive case for type application. The proof case for **error** was already discussed earlier, in Section 3.2. The only change necessary to what was said there is that instead of (3) we actually need to establish that $T(\llbracket \mathbf{error} \rrbracket_{\theta_1,\sigma_1} \;\$\$\; D_1) = T(\llbracket \mathbf{error} \rrbracket_{\theta_2,\sigma_2} \;\$\$\; D_2)$. But this is straightforward from the term semantics, which forces both tags to be Ok.

Another proof case already discussed in Section 3.2 is the one for the strict-let construct. Clearly, it also uses Lemma 2, to deduce $T(\llbracket t_1 \rrbracket_{\theta_1,\sigma_1}) = T(\llbracket t_1 \rrbracket_{\theta_2,\sigma_2})$ from $(\llbracket t_1 \rrbracket_{\theta_1,\sigma_1}, \llbracket t_1 \rrbracket_{\theta_2,\sigma_2}) \in \Delta^{err}_{\tau_1,\rho}$ and to deduce $(Bad\ (e_1 \cup E(\llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x \mapsto Bad\ \emptyset]})),$ $Bad\ (e_2 \cup E(\llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x \mapsto Bad\ \emptyset]}))) \in \Delta^{err}_{\tau_2,\rho}$ from $(Bad\ e_1, Bad\ e_2) \in \Delta^{err}_{\tau_1,\rho}$ and the statement that for every $(b, c) \in \Delta^{err}_{\tau_1,\rho}$, $(\llbracket t_2 \rrbracket_{\theta_1,\sigma_1[x \mapsto b]}, \llbracket t_2 \rrbracket_{\theta_2,\sigma_2[x \mapsto c]}) \in \Delta^{err}_{\tau_2,\rho}$.

Most other proof cases proceed like the corresponding ones for Theorem 1. The two that do not, and that require Lemma 2 as well, namely those for **case** and for type abstraction, are discussed in detail in the technical report [8].

3.4 Applying the New Parametricity Theorem

Having established Theorem 2, we can use it to derive free theorems that hold with respect to the imprecise error semantics. When doing so, we typically want to specialize relations (arising from the quantification in the definition of $\Delta^{err}_{\forall\alpha.\tau,\rho}$) to functions. To this end, the following definition is useful. The notation $\emptyset$ is used for empty mappings from type or term variables to elcpos and values.

Definition 3. *Let h be a term with $\vdash h : \tau_1 \to \tau_2$. The* graph *of h, denoted by $\mathcal{G}(h)$, is the relation $\{(a, b) \mid \llbracket h \rrbracket_{\emptyset,\emptyset} \;\$\; a = b\} \subseteq \llbracket \tau_1 \rrbracket_{\emptyset} \times \llbracket \tau_2 \rrbracket_{\emptyset}$. Note that it is actually a function, as h and a determine b.*

Of course, we should restrict attention to such h for which $\mathcal{G}(h)$ fulfills all necessary requirements on relations (i.e., error-strictness, continuity, and error-reflection). Error-strictness is easily translated from a restriction on $\mathcal{G}(h)$ to one on h. Continuity is a general property of functions and function application in the underlying semantics. Half of error-reflection, in the case of functions, is already given by error-strictness. The other half requires a guarantee that non-erroneous arguments are mapped to non-erroneous results. Altogether, we get the following definition and lemma.

Definition 4. *A term h with $\vdash h : \tau_1 \to \tau_2$ and $\llbracket h \rrbracket_{\emptyset,\emptyset} = Ok\ f$ is*

- error-strict *if $f\ a = a$ for every $a \in V_{err}$, and*
- error-total *if $T(f\ a) = Ok$ for every $a \in \llbracket \tau_1 \rrbracket_{\emptyset} \setminus V_{err}$.*

An h with $T(\llbracket h \rrbracket_{\emptyset,\emptyset}) = Bad$ is neither error-strict nor error-total.

For example, *null* is error-strict and error-total, while *tail* is neither. Also, Haskell's standard projection function *fst* is error-strict but not error-total, while (*const* 42) is error-total but not error-strict.

Lemma 3. *Let h be a term with $\vdash h : \tau_1 \to \tau_2$. Then $\mathcal{G}(h) \in \mathit{Rel}^{err}$ iff h is error-strict and error-total.*

We will only use the if-direction of this lemma, so we only sketch the proof of that direction, and only the parts related to the treatment of erroneous values:

- Error-strictness of $\mathcal{G}(h)$ follows from error-strictness of h by definition of \$.
- Error-reflection of $\mathcal{G}(h)$ follows from error-strictness and error-totality of h by the definition of \$ and because for every $a \in [\![\tau_1]\!]_\emptyset \setminus V_{err}$, $T(a) = \mathit{Ok}$ and $E(a) = \emptyset$, and for every $b \in [\![\tau_2]\!]_\emptyset$, $T(b) = \mathit{Ok}$ implies $E(b) = \emptyset$.

One further auxiliary lemma we need has to do with a connection between $\mathcal{G}$, the function *map*, and list^{err}.

Lemma 4. *Let h be a term with $\vdash h : \tau_1 \to \tau_2$. Then $\mathcal{G}(\mathit{map}\ \tau_1\ \tau_2\ h) = \mathit{list}^{err}\ \mathcal{G}(h)$.*

The proof (by coinduction, using the definition of list^{err} via a greatest fixpoint) holds no surprises and is thus omitted here.

We now have everything at hand to derive free theorems. For illustration, we take up the introductory example.

Example 1. Let t be a term with $\vdash t : \forall\alpha.(\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$. This requires to extend the calculus and proofs by integrating a Boolean type and associated term-formers with appropriate typing rules, semantics, and so on. Since the details are entirely straightforward, we omit them. By Theorem 2 we have $([\![t]\!]_{\emptyset,\emptyset}, [\![t]\!]_{\emptyset,\emptyset}) \in \Delta^{err}_{\forall\alpha.(\alpha\to\mathsf{Bool})\to[\alpha]\to[\alpha],\emptyset}$, where $\emptyset$ is now also used to denote the empty mapping from type variables to relations. Using the definition of Δ^{err}, we obtain that for every choice of elcpos D_1, D_2, relation $\mathcal{R} \in \mathit{Rel}^{err}(D_1, D_2)$, values p_1, p_2 with $(p_1, p_2) \in \Delta^{err}_{\alpha\to\mathsf{Bool},[\alpha\mapsto\mathcal{R}]}$, and l_1, l_2 with $(l_1, l_2) \in \mathit{list}^{err}\ \mathcal{R}$, $([\![t]\!]_{\emptyset,\emptyset}\ \$\$\ D_1\ \$\ p_1\ \$\ l_1, [\![t]\!]_{\emptyset,\emptyset}\ \$\$\ D_2\ \$\ p_2\ \$\ l_2) \in \mathit{list}^{err}\ \mathcal{R}$. Let h be a term with $\vdash h : \tau_1 \to \tau_2$ that is error-strict and error-total. (For the introductory example, $h = \mathit{tail}$ is neither error-strict nor error-total.) By Lemma 3 we have $\mathcal{G}(h) \in \mathit{Rel}^{err}([\![\tau_1]\!]_\emptyset, [\![\tau_2]\!]_\emptyset)$, so we can use it to instantiate $\mathcal{R}$ above. By Lemma 4 we then have $\mathit{list}^{err}\ \mathcal{R} = \mathcal{G}(\mathit{map}\ \tau_1\ \tau_2\ h)$, and thus for every choice of values p_1, p_2 with $(p_1, p_2) \in \Delta^{err}_{\alpha\to\mathsf{Bool},[\alpha\mapsto\mathcal{G}(h)]}$ and $l_1 \in [\![[\tau_1]]\!]_\emptyset$, $[\![\mathit{map}\ \tau_1\ \tau_2\ h]\!]_{\emptyset,\emptyset}\ \$\ ([\![t\ \tau_1]\!]_{\emptyset,\emptyset}\ \$\ p_1\ \$\ l_1) = [\![t\ \tau_2]\!]_{\emptyset,\emptyset}\ \$\ p_2\ \$\ ([\![\mathit{map}\ \tau_1\ \tau_2\ h]\!]_{\emptyset,\emptyset}\ \$\ l_1)$. The condition on p_1 and p_2 unfolds to $T(p_1) = T(p_2)$ and for every $a \in [\![\tau_1]\!]_\emptyset$, $p_1\ \$\ a = p_2\ \$\ ([\![h]\!]_{\emptyset,\emptyset}\ \$\ a)$. The latter is easy to satisfy by choosing $p_1 = [\![\lambda x : \tau_1.p\ (h\ x)]\!]_{\emptyset,\emptyset}$ and $p_2 = [\![p]\!]_{\emptyset,\emptyset}$ for some p with $\vdash p : \tau_2 \to \mathsf{Bool}$, but we need to take note of the requirement that $T([\![\lambda x : \tau_1.p\ (h\ x)]\!]_{\emptyset,\emptyset}) = T([\![p]\!]_{\emptyset,\emptyset})$, which is equivalent to $T([\![p]\!]_{\emptyset,\emptyset}) = \mathit{Ok}$. Altogether, we now have for every $l_1 \in [\![[\tau_1]]\!]_\emptyset$, $[\![\mathit{map}\ \tau_1\ \tau_2\ h]\!]_{\emptyset,\emptyset}\ \$\ ([\![t\ \tau_1\ (\lambda x : \tau_1.p\ (h\ x))]\!]_{\emptyset,\emptyset}\ \$\ l_1) = [\![t\ \tau_2\ p]\!]_{\emptyset,\emptyset}\ \$\ ([\![\mathit{map}\ \tau_1\ \tau_2\ h]\!]_{\emptyset,\emptyset}\ \$\ l_1)$, and thus for every term l with $\vdash l : [\tau_1]$, $[\![\mathit{map}\ \tau_1\ \tau_2\ h\ (t\ \tau_1\ (\lambda x : \tau_1.p\ (h\ x))\ l)]\!]_{\emptyset,\emptyset} = [\![t\ \tau_2\ p\ (\mathit{map}\ \tau_1\ \tau_2\ h\ l)]\!]_{\emptyset,\emptyset}$ under the conditions that h is error-strict and error-total and that $T([\![p]\!]_{\emptyset,\emptyset}) = \mathit{Ok}$. This is the promised equivalence repair for (1).

4 Dealing with Exceptions, and Beyond

So far, we have dealt with errors as events that lead a program to fail, without any possibility to manipulate them from inside the language itself, or to even recover from them. The accompanying technical report [8] shows how to deal with the Haskell primitive *mapException*, also discussed by Peyton Jones et al. [5]. Eventually, we want to cover full exception handling by (partially) modeling Haskell's IO monad.

An important move towards practical applicability would be the provision of static analyses that can successfully check for the conditions under which free theorems are now known to hold, so as to make them safely usable in a Haskell compiler. Clearly, conditions like error-strictness and error-totality are undecidable in general. But considering slightly stronger, tractable, conditions could be good enough in practice. In particular, it should be possible to leverage GHC's strictness analyzer for also establishing error-strictness, and a sufficient check for error-totality is possible using the strategy employed by Xu et al. [10], namely symbolic evaluation plus syntactic safety, all ready for the taking in (a branch of) GHC.

Acknowledgements. We would like to thank anonymous reviewers for their comments and suggestions.

References

1. P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *POPL, Proceedings*, pages 99–110. ACM Press, 2004.
2. P. Johann and J. Voigtländer. A family of syntactic logical relations for the semantics of Haskell-like languages. *Information and Computation*, 207(2):341–368, 2009.
3. A. Moran, S.B. Lassen, and S.L. Peyton Jones. Imprecise exceptions, Co-inductively. In *HOOTS, Proceedings*, volume 26 of *ENTCS*, pages 122–141. Elsevier, 1999.
4. S.L. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Marktoberdorf Summer School 2000, Proceedings*, pages 47–96. IOS Press, 2001.
5. S.L. Peyton Jones, A. Reid, C.A.R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *PLDI, Proceedings*, pages 25–36. ACM Press, 1999.
6. A.M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.
7. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
8. F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. Technical Report TUD-FI08-08, Technische Universität Dresden, 2008.
9. P. Wadler. Theorems for free! In *FPCA, Proceedings*, pages 347–359. ACM Press, 1989.
10. D.N. Xu, S.L. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *POPL, Proceedings*, pages 41–52. ACM Press, 2009.