Type Error Customization in GHC

Controlling expression-level type errors by type-level programming

Alejandro Serrano
A.SerranoMena@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Jurriaan Hage
J.Hage@uu.nl
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

ABSTRACT

Embedded domain specific languages (DSLs) are a common pattern in the functional programming world, providing very high-level abstractions to programmer. Unfortunately, this abstraction is broken when type errors occur, leaking details of the DSL implementation. In this paper we present a set of techniques for customizing type error diagnosis in order to avoid this leaking. These techniques have been implemented in the GHC Haskell compiler.

Our customizations are declared in the type signatures of functions provided by the DSL, leading to type error message that are context-dependent: the same kind of error can be reported in a different way depending on the particular expression in which it occurs. We make use of the ability to manipulate constraints using type-level programming which is already present in GHC, and which enables reuse and abstraction of common type error patterns.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**;
• **Theory of computation → Type structures**;

ACM Reference Format:
Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling expression-level type errors by type-level programming. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3205368.3205370

1 INTRODUCTION

Domain Specific Languages (or DSLs for short) are a widely used technique. Their main advantage is the effectiveness with which domain experts can describe the solution to a problem, even with little prior programming experience. Examples abound, ranging from SQL for database processing to describing drawings [Yorgey 2012] and even music harmony analysis [Koops et al. 2013]. There are two main approaches to developing DSLs: in the *external* approach a new compiler toolchain is built, including a parser, a code generator and associated tooling. The drawback of such a standalone DSL is the amount of work required; for that reason frameworks have been designed to help developers in this task [Voelter 2013].

The second approach is that of *internal* or *embedded* DSLs [Hudak 1996]. Embedded DSLs are developed as libraries in a host language, in order to reuse all the machinery which is already implemented for that host. Another advantage of embedded DSLs is the ease with which DSLs can be combines in a single application. The embedded approach is common in the functional programming world and is the focus of this paper. From now on, when we talk of DSLs we mean embedded DSLs.

DSLs provide a very high level of abstraction to the programmer, who communicates with the compiler using domain terms. Alas, this abstraction is broken when a type error occurs. From the point of view of the compiler, a DSL is merely a library, and by itself cannot phrase error messages in domain terms. The result is that errors leak details of the implementation of the DSL to the user, a serious disadvantage [Hage 2014]. Some authors of DSLs include a special section in the documentation devoted to explain the most common type errors, like in `diagrams` [Yorgey 2016, section 5.5], whereas others like `persistent` [Snoyman 2012] directly document the library internals. Both solutions are clearly suboptimal.

This problem has been acknowledged by several authors, who have proposed various mechanisms for DSL authors to customize the errors generated by the compiler [Christiansen 2014; Heeren et al. 2003a; Plociniczak et al. 2014; Serrano and Hage 2016b; Wazny 2006], so that type error messsages can be phrased in terms of the domain instead of the underlying encoding of that domain in the host language. This paper introduces type error customization in GHC, the main implementation of the Haskell language [Marlow et al. 2010], showing that powerful customized type error diagnosis is available for a language as large and complex as Haskell.

1.1 Type Errors

Before we describe our solution, we need to understand the different sources of errors during type checking. We stress that our focus is on type errors, so we assume that code has already been successfully parsed and names have already been resolved; errors may only arise from problems with typings.

Earlier in this paper, we mentioned `diagrams`, a DSL for representing drawings and animations. The fundamental type in this library is *QDiagram b v n m*, where the type parameters represent, in order, the drawing back-end – such as SVG or bitmap–, the vector space – `diagrams` deals with 2-D and 3-D images in a uniform way

This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213).

IFL 2017, August 30-September 1, 2017, Bristol, United Kingdom

ACM ISBN 978-1-4503-6343-3/17/08…$15.00
https://doi.org/10.1145/3205368.3205370

–, the type of numeric coordinates, and the type used for annotating the diagrams. In most cases, we can only combine diagrams when all of these type parameters coincide. One such example is the *atop* combinator used to arrange diagrams vertically:

$$\begin{aligned} atop \;::\; & (OrderedField\ n, Metric\ v, Semigroup\ m) \\ & \Rightarrow QDiagram\ b\ v\ n\ m \to QDiagram\ b\ v\ n\ m \\ & \to QDiagram\ b\ v\ n\ m \end{aligned}$$

The first group of type errors consists of those arising from *inconsistencies*. In this case, an expression in the program is expected to have two different incompatible types. A simple example is when we pass to *atop* the arguments *True* and `'c'`:

```
Couldn't match type 'QDiagram b v n m' with type 'Bool'
```

Inconsistencies might also arise at a deeper level. For example, the arguments to *atop* might be rightful diagrams, but living in different vector spaces. In that case, the default error message:

```
Couldn't match type 'V2' with type 'V3'
```

already assumes some knowledge of the internals of the library, namely that *V2* and *V3* are the types used to represent the two- and three-dimensional vector spaces, and that this information is encoded in the diagram type. A better diagnosis uses domain terms:

```
Vector spaces do not coincide: V2 vs. V3
```

Since version 8.0, users of GHC have the ability to produce custom inconsistency errors during type resolution by means of a special construct *TypeError* [Diatchki 2015]. The archetypal example is providing a better user experience when using the *show* function which converts a value into its string representation. In the general case, we do not want to allow functions to appear as arguments to *show*. The solution is to write an instance of the *Show* type class which has one *TypeError* constraint as context:

$$\begin{aligned} & \textbf{instance}\ TypeError\ (Text\ \texttt{"Functions cannot be shown"}) \\ & \qquad \Rightarrow Show\ (a \to b) \end{aligned}$$

Now, if the user tries to compile $show\ (\lambda x \to x)$, he or she receives that custom message instead of the more cryptic[1]

```
No instance for (Show (t0 -> t0))
arising from a use of 'show'
```

Haskell, among other languages, supports ad-hoc overloading via type classes. Functions can require an instance of a type class for the specific type the polymorphic function is applied to. The *atop* function introduced previously uses such a mechanism: it requires the vector space v to have a *Metric* instance, the number field n must be ordered and the annotation type m must form a semigroup. Operationally, each type class requirement is transformed into an extra argument which is filled in automatically by the compiler as part of the typing process.

This mechanism gives rise to a second source of errors: an instance of a given type class is not available for a specific type. This is the case in the previous example where a function cannot be shown. Given the operational view we have just explained, it makes sense for the compiler to signal an error, since without an instance there is no way to fill in the extra argument coming from the type class requirement. We note that programs written in languages featuring implicit parameter resolution – such as Scala, Agda, Idris, and Coq – can also exhibit this kind of error.

If you think of type checking as a process where constraints between types are generated and then solved – an intuition we shall formalize in just a moment –, these errors come from constraints which are *left undischarged*.

As with inconsistency errors, left-undischarged errors are also amenable to customization. One good example is `persistent`, a type-safe database library developed for Haskell. When using this library, each data type which can be persisted needs to be annotated in a certain way [Snoyman 2012], resulting in an instance of the *PersistEntity* type class. This requirement is visible in the types of the query and update operations:

$$\begin{aligned} & insertUnique \\ & \quad :: (MonadIO\ m, PersistUniqueWrite\ backend, PersistEntity\ record) \\ & \Rightarrow record \to ReaderT\ backend\ m\ (Maybe\ (Key\ record)) \end{aligned}$$

If we find out that an instance of *PersistEntity* is missing, we can do better than plainly reporting this fact: we can point the user to the documentation discussing how to properly annotate the data type. Even more, the user need not be aware of the fact that type classes are involved in the implementation, we can just output:

```
Data type 'Person' is not declared as an entity.
Hint: entity definition can be automatically derived.
Check the documentation at http://www.yesodweb.com/...
```

We note that the implicit parameter system in Scala has some support for customization in the form of the *@implicitNotFound* annotation [Scala Team 2015]. Whenever an implicit value cannot be inferred for a given trait, the declared message is shown instead of the compiler's default one.

$$\begin{aligned} & @implicitNotFound\ (msg = \texttt{"Data type is not an entity"}) \\ & \textbf{trait}\ PersistEntity\ [A]\ \{\ldots\} \end{aligned}$$

A third kind of error which may arise during type checking is *ambiguity* errors. Ambiguous types are reported when for a given type variable the compiler is unable to find a ground type to unify with, nor is it able to quantify over the type variable. In our own experience, these problems do not often call for a domain-specific type error diagnosis solution, but rather for more generic techniques such as *defaulting* [Heeren and Hage 2005; Marlow et al. 2010]. For that reason, we consider in this paper only inconsistencies and left-undischarged errors.

1.2 Context-Dependent Errors

We have discussed how several languages provide support for customizing left-undischarged errors and generate inconsistency errors with a custom message. Is this enough to provide good error messages for embedded DSLs? Not at all! When using a DSL, the context in which an expression appears gives meaningful information which should be considered when building an error message.

As an example, let us consider a version of *atop* in which each single constraint over the types appears as a single constraint in the source code. In particular, instead of a single equality constraint between the arguments,

$$QDiagram\ b_1\ v_1\ n_1\ m_1 \sim QDiagram\ b_2\ v_2\ n_2\ m_2$$

[1] In newer versions of GHC, that error message is extended with a hint (`maybe you haven't applied a function to enough arguments?`). However, this special reporting is baked into the compiler, DSL authors are not able to provide such hints.

we indicate the equalities between the sub-components. Note that we follow GHC's convention of denoting the equality between types t_1 and t_2 as $t_1 \sim t_2$.

$$\begin{aligned} \mathit{atop} \;::\; & (d_1 \sim \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1, \\ & d_2 \sim \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2, \\ & b_1 \sim b_2, v_1 \sim v_2, n_1 \sim n_2, m_1 \sim m_2, \\ & \mathit{OrderedField}\ n_1, \mathit{Metric}\ v_1, \mathit{Semigroup}\ m_1) \\ \Rightarrow\ & d_1 \to d_2 \to d_1 \quad \text{-- Argument types replaced by variables} \end{aligned}$$

From the compiler's point of view, the equality $b_1 \sim b_2$ is no different from $v_1 \sim v_2$. But from the DSL point of view, these are quite different: the former checks an agreement between back-ends, whereas the latter speaks about vector spaces. If an inconsistency is found between values of v_1 and v_2 we want to report:

```
Vector spaces do not coincide: V2 vs. V3
```

It is not only the position of a variable in a longer type, what custom message to report also depends on the function being used. Consider the following combinator:

$$\begin{aligned} \mathit{strokePath} \;::\; & (\mathit{TypeableFloat}\ n, \mathit{Renderable}\ (\mathit{Path}\ \mathit{V2}\ n)\ b) \\ \Rightarrow\ & \mathit{Path}\ \mathit{V2}\ n \to \mathit{QDiagram}\ b\ \mathit{V2}\ n\ \mathit{Any} \end{aligned}$$

In this case we demand that the argument lives in a two-dimensional space by setting the corresponding type parameters to *V2*. Whereas in the *atop* case there are two vector spaces with equal status, in the case of *strokePath* the two-dimensional requirement is key, and thus we prefer a message along the lines of:

```
'strokePath' requires a V2 path, but was passed a V3
```

The dependence of the wording of the type error message on the context is something we cannot achieve with any of the previous solutions. *TypeError*s can generate new errors or replace left-undischarged ones with inconsistencies, but only reflecting on information at the type level. In a similar fashion, *@implicitNotFound* belongs to a particular trait; the same custom message is reported regardless of where the trait is required.

1.3 Contributions

In this paper we present two mechanisms for context-dependent type error customization. The first technique (§ 3) is very simple to implement, but has some limitations with respect to ordering the checks done inside the compiler. In § 4 we present another technique which lifts these limitation.

Both techniques have several advantages compared to previous proposals on type error customization:

- They do not require changes to libraries: a wrapper providing better error messages can be developed independently. This also opens the door to distributing several versions of the same API whose error messages target specific users.
- Error customizations blend with other type-level constructs. That means that we can use the type-level facilities in GHC, such as type families, to provide facilities for abstraction and reuse when implementing our customizations.
- Custom messages become part of the exported type interface, not part of a document which users might not be aware of or may become outdated.

We have implemented both techniques inside of GHC, and developed a library of useful combinators targeting common type error patterns (§ 5). Using this library, we have wrapped some popular libraries like the aforementioned `diagrams` and `persistent`. An example of the result is given in Figure 1 (:≁: represents inequality). Note that we can piggy-back on the *Diagrams.Combinators* version in the implementation. In Appendix B we fully describe the annotations needed for the `path` library.

2 CONSTRAINT-BASED TYPE CHECKING

Type checking in GHC, starting with version 7, is based on the OutsideIn(X) system [Vytiniotis et al. 2011]. OutsideIn(X) belongs to the family of type systems whose operation is based on constraints; this also includes other functional languages such as ML [Pottier and Rémy 2005] and Swift [Swift Team 2016]. OutsideIn(X) is itself generic, and is instantiated for a constraint system X. In the specific case of GHC, X includes type classes and type families.

Constraint-based type checkers perform their work in two phases. The communication between phases is mediated via *constraints*, which are specific to each type system, although almost all of them include some form of type equality. In the case of GHC, constraints conform to the following syntax:

Primitive constraints	P	::=	$\tau_1 \sim \tau_2 \mid \mathsf{C}\ \tau_1\ \ldots\ \tau_n$
Simple constraints	Q	::=	$\varepsilon \mid Q_1, Q_2 \mid P$
(Extended) constraints	C	::=	$Q \mid C_1, C_2 \mid \exists \overline{\alpha}.(Q \supset C)$

Primitive constraints include type class instance constraints, such as *Show* [*Int*], in addition to equalities. Implication constraints are needed to model pattern matching over generalized algebraic data types (GADTs), where different branches refine the information we have about types. These are called extended constraints in OutsideIn(X); for conciseness we shall drop the adjective. At this point, the addition of the level of simple constraints, including only conjunction, in between primitive and normal ones is not standard; the reason for introducing it shall become apparent later.

The first phase in the type checking process, usually called *generation* or *gathering*, traverses the Abstract Syntax Tree of the program and produces a set of constraints which the types of the expressions need to satisfy. In some cases, this traversal also involves the propagation of some information, like pushing type signatures towards the leaves of the tree. In the case of OutsideIn(X), generation is represented by a judgment $\Gamma \vdash e : \tau \rightsquigarrow C$ which states that under environment Γ the expression e is assigned a type τ given that the constraint set C is satisfied. We refrain from describing the entire judgment, the fragment given in Figure 2 is enough for our purposes. The syntax $[\overline{a \mapsto \alpha}]$ represents the substitution of the rigid variables in the type signature of x by fresh type variables.

After constraints have been gathered we need to *solve* them. From a high-level point of view solving is described by a judgment $\mathcal{Q} \vdash C_w \rightsquigarrow Q_r; \theta$ stating that under a set of axioms $\mathcal{Q}$ the wanted constraints C_w can be simplified to a set of simple constraints Q_r and a type assignment θ. In this case, axioms refer to global information like the declared instances for every type class. For example, if we take $\mathcal{Q}$ to contain the following declarations:

class $\mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a$ **instance** $\mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]$

```
atop :: CustomErrors [
  [d1 :⁄∼: QDiagram b1 v1 n1 m1 :⇛: Text "Arg. #1 to 'atop' must be a diagram",
   d2 :⁄∼: QDiagram b2 v2 n2 m2 :⇛: Text "Arg. #2 to 'atop' must be a diagram"],
  [DoNotCoincide "Back-ends" b1 b2, DoNotCoincide "Vector spaces" v1 v2,
   DoNotCoincide "Numerical fields" n1 n2, DoNotCoincide "Query annotations" m1 m2],
  [Check (OrderedField n1), Check (Metric v1), Check (Semigroup m1)]
 ] ⇒ d1 → d2 → d1
atop = Diagrams.Combinators.atop
```

Figure 1: Type signature for *atop* with custom error diagnosis

$$\frac{\overline{\alpha}\text{ fresh} \qquad (x : \forall \overline{a}.Q \Rightarrow \tau) \in \Gamma}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q}$$

$$\frac{\alpha\text{ fresh} \qquad \Gamma, (x : \alpha) \vdash e : \tau \rightsquigarrow C}{\Gamma \vdash (\lambda x \to e) : \alpha \to \tau \rightsquigarrow C}$$

$$\frac{\Gamma \vdash f : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e : \tau_2 \rightsquigarrow C_2 \qquad \alpha\text{ fresh}}{\Gamma \vdash f\ e : \alpha \rightsquigarrow C_1, C_2, \tau_1 \sim (\tau_2 \to \alpha)}$$

Figure 2: Constraint generation for a subset of Haskell

Then we have that $Q \vdash Ord\ a, Eq\ [a] \rightsquigarrow Ord\ a$, since $Eq\ [a]$ is simplified to $Eq\ a$, and we know that this second constraint holds because we have $Ord\ a$.

Under the hood, the solver consists of a driver which takes care of implications and a rewriting procedure working over simple constraints. In a simplified form, rewriting is expressed as a judgment $Q \vdash \langle \varphi; Q_a; Q_w \rangle \hookrightarrow \langle \varphi'; Q'_a; Q'_w \rangle$. The set of axioms does not change throughout the process, but the ongoing substitution φ, the assumed constraints Q_a – coming from the implications gathered from pattern matching over a GADT – and the wanted constraints Q_w do. Each rewrite step applies a single simplification to the constraint sets, like splitting a type equality headed by a constructor into type equalities on its arguments:

$$\begin{array}{lll} Q & \vdash & \langle \varphi\ ;\ (\mathsf{T}\,\tau_1\ \ldots\ \tau_n \sim \mathsf{T}\,\rho_1\ \ldots\ \rho_n), Q_a\ ;\ Q_w \rangle \\ & \hookrightarrow & \langle \varphi\ ;\ \tau_1 \sim \rho_1, \ldots, \tau_n \sim \rho_n, Q_a\ ;\ Q_w \rangle \\ Q & \vdash & \langle \varphi\ ;\ Q_a\ ;\ (\mathsf{T}\,\tau_1\ \ldots\ \tau_n \sim \mathsf{T}\,\rho_1\ \ldots\ \rho_n), Q_w \rangle \\ & \hookrightarrow & \langle \varphi\ ;\ Q_a\ ;\ \tau_1 \sim \rho_1, \ldots, \tau_n \sim \rho_n, Q_w \rangle \end{array}$$

At each of these steps the solver might detect an inconsistency, such as $Int \sim Bool$. In that case, a special constraint $\bot$ is generated.

This description of solving allows us to formally define what the two kinds of errors. *Inconsistencies* occur when a constraint has been rewritten to $\bot$. Thus, what is considered an inconsistency depends on the type system at hand. In a real implementation, each of these $\bot$ constraints keeps track of the constraints it came from, in order to report the corresponding error message. *Left-undischarged errors* are reported whenever the set of residual constraints was expected to be empty – like when checking a function definition against its type signature – but this was not the case. In most cases, in the presence of inconsistencies left-undischarged constraints are not reported.

2.1 The *Constraint* kind

Starting with version 7.4, GHC considers constraints as types of the special kind *Constraint* [Bolingbroke 2011]. This implies that all the constructs available for type-level programming can be used also to manipulate constraints. The simplest example is giving an alias to a set of constraints. If we have the following declaration:

```
type JSONSerializable a = (FromJSON a, ToJSON a)
```

we can write *JSONSerializable Person* instead of both *FromJSON Person* and *ToJSON Person*. The kind of *JSONSerializable* is inferred by the compiler to be $* \to Constraint$, where $*$ is the kind of ground types.

More interesting is the combination of the *Constraint* kind and type families, a form of type-level functions in GHC. The following example applies a type class to all the elements of a type-level list:

```
type family All (c :: k → Constraint) (xs :: [k]) where
  All c []       = ()
  All c (x : xs) = (c x, All c xs)
```

This way, *All Show* [*Int*, *Bool*] is equivalent to (*Show Int*, *Show Bool*). For the previously introduced *TypeError*, we can use type families to abstract common patterns in those cases. For example, it is very common that a type class is not implementable for functions. We can write a synonym for that error. Here we use the combinator `:⋄:` to concatenate two messages, and the special type family *ShowType* which the compiler turns into a text representation of the type given as argument.

```
type NoFunctionsAllowed c
  = TypeError (Text "Cannot " :⋄: ShowType c :⋄: " functions")
```

which we can later use in the definition of the instance for functions:

```
instance NoFunctionsAllowed Show ⇒ Show (a → b)
```

In that way, we ensure that error messages are all built from the same textual template, instead of each specific instance accidentally having a slightly different wording. This facility opens the way to error customization which can be reused and manipulated: by ensuring that our customizations are expressed as *Constraint*s.

3 HINTING AT SOLUTIONS

In the first technique we present we annotate constraints with messages which ought to be reported when the constraint leads to an inconsistency or is left undischarged. Using this mechanism, the type signature of the *atop* function from `diagrams` is annotated as shown in Figure 3. We remark that this type signature is plain Haskell (with some GHC extensions enabled). The *InconsistentHint*

annotations are predicates which influence the way type error messages are reported.

If we now try to call the function with values of the wrong type, the custom text is attached as a hint to the standard error:

example = *atop True* `'c'`

```
* Couldn't match type 'QDiagram b v n m' with type 'Bool'
* In the expression: atop True 'c'
* Hint: argument #1 to 'atop' must be a diagram
```

The second argument to *InconsistentHint* must be of kind *ErrorMessage*, the same kind as that of *TypeError*. Therefore, our hint is able to show the types involved in the inconsistency. We could refine the third hint to include the names of the offending back-ends:

$(b_1 \sim b_2)$ `` `InconsistentHint` ``
(*Text* `"the diagrams use different back-ends "` :⋄: *ShowType* b_1 :⋄: *Text* `" and "` :⋄: *ShowType* b_2)

We stress that hints are attached to a specific constraint in a specific type signature, achieving the *context-dependency* we were aiming for. These hints do not need to appear in the original DSL, we can create a wrapper library after the fact. For example, we can make *atop* point to the original `diagrams` version:

import qualified *Diagrams.Combinators*

atop :: (...) ⇒ $d_1 \to d_2 \to d_1$ -- Previous annotated signature
atop = *Diagrams.Combinators.atop*

Furthermore, by simply type checking this definition, the compiler ensures that the annotated type signature is compatible with the original type. Checking this soundness property requires additional work in other error customization proposals [Heeren et al. 2003a; Serrano and Hage 2017], while in our case it follows from the rewriting rules of hint combinators.

For left-undischarged errors, we provide *LeftUndischargedHint*, allowing us to point to the derivation facilities in `persistent`, as shown in Figure 4. Now, if the type of the value we try to insert in the database is not declared as an entity in Persistent terms, the user obtains a suggestion on how to fix the problem. Using this same idea, we can also improve the user experience for type classes which can be automatically derived, such as *Eq* and *Ord*.

3.1 Implementation

The implementation of hints inside GHC is divided into two parts: the user-facing library part and some modifications to the type checker itself. The former part consists of definitions of new constraint combinators:

class $c \Rightarrow$ *InconsistentHint* c (*msg* :: *ErrorMessage*)
instance $c \Rightarrow$ *InconsistentHint* c *msg*

class $c \Rightarrow$ *LeftUndischargedHint* c (*msg* :: *ErrorMessage*)
instance $c \Rightarrow$ *LeftUndischargedHint* c *msg*

These instances imply that to solve either c `` `InconsistentHint` `` *msg* or c `` `LeftUndischargedHint` `` *msg* the compiler must solve c itself. Note that this definition, parametric in the constraint c, would not have been possible before the introduction of the *Constraint* kind.

Using type classes to recall the hints poses in principle a performance problem, since dictionaries have to be created and passed around. We can solve this problem by asking the compiler to *inline* the definition of the hint-annotated version.

```
{-# INLINE atop #-}
```

Now, after type checking, each call to *atop* is replaced by its body, *Diagrams.Combinators.atop*, making the use of the annotated version essentially for free from a performance perspective.

To make sure our annotations are taken into account, we need to change the solving process inside the compiler to retain information about our hints. In the OutsideIn(X) formalism, we modify the syntax of constraints. Now simple constraints are not merely conjunctions of primitive constraints, but each of these primitive constraints is optionally annotated with hints. A constraint Q annotated with a inconsistent hint $h_\perp$ and a left-undischarged hint $h_{\mathcal{U}}$ is written as $Q^{h_\perp / h_{\mathcal{U}}}$. The grammar of such hints is just a copy of GHC's *ErrorMessage* kind.

Simple constraints	Q	::=	$\varepsilon \mid Q_1, Q_2 \mid P \mid Q^{h_\perp / h_{\mathcal{U}}}$
Hints	h	::=	– (No hint)
		|	*Text* `"t"` | *ShowType* τ | h_1 :⋄: h_2

Hints are updated by the solver whenever a qualifier for our specials classes *LeftUndischargedHint* and *InconsistentHint* are encountered. Errors may only appear with constraints we aim to *solve*, not from those coming from the antecedent of an implication. Thus, we only need to do special handling when the type class predicates arise in a wanted position. In terms of our rewriting judgment:

$$\begin{array}{lll} Q & \vdash & \langle \varphi \ ; \ Q_a \ ; (Q \text{ `InconsistentHint` } h^{\star}_{\perp})^{h_\perp / h_{\mathcal{U}}}, Q_w \rangle \\ & \hookrightarrow & \langle \varphi ; Q_a ; Q^{h^{\star}_{\perp} / h_{\mathcal{U}}}, Q_w \rangle \\ Q & \vdash & \langle \varphi \ ; \ Q_a \ ; (Q \text{ `LeftUndischargedHint` } h^{\star}_{\mathcal{U}})^{h_\perp / h_{\mathcal{U}}}, Q_w \rangle \\ & \hookrightarrow & \langle \varphi ; Q_a ; Q^{h_\perp / h^{\star}_{\mathcal{U}}}, Q_w \rangle \end{array}$$

Every rewriting rule in the solver must be updated to propagate hints, otherwise hints attached to constraints that only indirectly turn into an inconsistency will not be reported. As an example, this is the new rule for splitting type equalities:

$$\begin{array}{lll} Q & \vdash & \langle \varphi ; Q_a ; (\mathsf{T}\ \tau_1 \ \ldots \ \tau_n \sim \mathsf{T}\ \rho_1 \ \ldots \ \rho_n)^{h_\perp / h_{\mathcal{U}}}, Q_w \rangle \\ & \hookrightarrow & \langle \varphi ; Q_a ; (\tau_1 \sim \rho_1)^{h_\perp / h_{\mathcal{U}}}, \ldots, (\tau_n \sim \rho_n)^{h_\perp / h_{\mathcal{U}}}, Q_w \rangle \end{array}$$

In general, this propagation strategy becomes problematic when more than one constraint is involved in a rewriting step [Serrano and Hage 2016a], since the solver may have to merge different hints.

In the GHC type checker there is the notion of *work item*, which is the constraint considered at each point in time. A work item goes through different phases, including canonicalization and reaction with constraints in the so-called inert set. We made the design decision of propagating always the hints from the work item from any new constraints derived from an interaction with other constraints. This change is also the least invasive for the GHC code base, and works well in practice.

3.2 Reuse and Abstraction

In § 2.1, we showed how by means of a type synonym of kind *Constraint* we can reuse messages in *TypeError*. The same technique can be applied to our hints. One common use case is exporting a synonym which comes with the default annotation, instead of the

```
atop :: ((d1 ~ QDiagram b1 v1 n1 m1) `InconsistentHint` (Text "argument #1 to 'atop' must be a diagram"),
         (d2 ~ QDiagram b2 v2 n2 m2) `InconsistentHint` (Text "argument #2 to 'atop' must be a diagram"),
         (b1 ~ b2)   `InconsistentHint` (Text "the diagrams must use the same back-end"),
         (v1 ~ v2)   `InconsistentHint` (Text "the diagrams must live in the same vector space"),
         (n1 ~ n2)   `InconsistentHint` (Text "the diagrams must use the same numeric field"),
         (m1 ~ m2) `InconsistentHint` (Text "the diagrams use different annotations"),
         OrderedField n1, Metric v1, Semigroup m1) => d1 -> d2 -> d1
```

Figure 3: Annotated type signature for *atop* with hints

```
insertUnique :: (MonadIO m, PersistUniqueWrite backend,
                 PersistEntity record `LeftUndischargedHint` (
                   Text "Data type '" :<>: ShowType record :<>: Text "' is not declared as entity."
                   :$$: Text "Hint: entity definition can be automatically derived."
                   :$$: Text "Check the documentation at http://www.yesodweb.com/...")
                 => record -> ReaderT backend m (Maybe (Key record))
```

Figure 4: Annotated type signature for *insertUnique*

bare type class. For example, the authors of `persistent` could re-define *PersistEntity* as follows:

```
type PersistEntity r = PersistEntity' r `LeftUndischargedHint`
  (Text "Data type '" :<>: ShowType r :<>: ...)
```

where we assume the old *PersistEntity* from the Persistent library has been renamed to *PersistEntity′*. In that way, every time an expression mentions the *PersistEntity* type class in a type signature, the hint comes attached to it and will be shown to the DSL user if the constraint is left undischarged.

We can go one step further by *abstracting* over common patterns in DSL type error diagnosis. As usual in functional programming, this abstraction is described by a function; type families are the tool for type-level functions in Haskell.

Our annotated type signature for *atop* at the beginning of the section has some clear duplication. First, for both arguments we check that they have the correct head constructor, *QDiagram*. Then, for every argument to *QDiagram*, we check for equality between that component, with a similar *InconsistentHint* for each case. We can remove the apparent duplication by introducing some combinators, resulting in the cleaner signature given in Figure 5.

The definitions of *ShapeHints* and *ArgsHints* are given in Figure 6. In both cases we traverse a list of tuples containing the elements which should be equal and information needed to produce the hint if they are not. In the case of *ShapeHints* we use a counter which is incremented at every step to be able to inform which argument did not have the right shape. For the hints over arguments, we also require a name for the whole structure; such type-level string literals have the kind *Symbol* in GHC.

4 CONTROLLING THE SOLVING ORDER

The technique of adding hints to constraints as described in § 3 is simple and straightforward to implement. Unfortunately, it does not act in a predictable way due to the unspecified order of constraint solving. In this section we describe the problems and a new constraint combinator, *IfNot*, which solves these problems.

Let us look at a concrete example in which we obtain a surprising hint for our *atop* example. Say we want to combine two diagrams $d_1 ::$ *QDiagram SVG V2 Double* () and $d_2 ::$ *QDiagram SVG V3 Double* (). The difference between the types lies in the vector space component, so we expect the hint to refer to this fact. The first step in type checking *atop* d_1 d_2 is gathering the necessary constraints. After some simplifications, the constraint set contains the equalities given below; we identify each one by the number on top of the $\sim$ symbol. Furthermore, each constraint is annotated with an inconsistency hint coming from the *atop* signature, which we refrain from showing here for the sake of conciseness.

$$\textit{QDiagram SVG V2 Double}\ () \overset{1}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1,$$
$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_2\ v_2\ n_2\ m_2,$$
$$b_1 \overset{3}{\sim} b_2, v_1 \overset{4}{\sim} v_2, n_1 \overset{5}{\sim} n_2, m_1 \overset{6}{\sim} m_2$$

Since the constraint solver is allowed to consider equalities in any order, it might well perform the four last substitutions first. We are then left with the following two constraints:

$$\textit{QDiagram SVG V2 Double}\ () \overset{1}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1,$$
$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_1\ v_1\ n_1\ m_1$$

From the first constraint the solver now discovers that $v_1 \sim \textit{V2}$, and applies this fact in the second constraint:

$$\textit{QDiagram SVG V3 Double}\ () \overset{2}{\sim} \textit{QDiagram}\ b_1\ \textit{V2}\ n_1\ m_1$$

The problem to report is that *V2* does not match *V3*. But the hint associated with the second constraint is `argument #2 to 'atop' must be a diagram`. Reporting that hint as part of the error message is completely misleading for the user: the second argument is in fact a diagram!

Our solution is to give some control to the DSL author over the order in which constraints are considered. If we ensure that the $d \sim \textit{QDiagram}\ b\ v\ n\ m$ constraints are solved before the others, the misleading hint will never arise. Techniques for custom ordering of constraints are available in the Helium Haskell compiler [Hage

$$atop :: (ShapeHints\ [(d_1, QDiagram\ b_1\ v_1\ n_1\ m_1, \texttt{"diagram"}), (d_2, QDiagram\ b_2\ v_2\ n_2\ m_2, \texttt{"diagram"})],$$
$$ArgsHints\ \texttt{"diagrams"}\ [(b_1, b_2, \texttt{"back-ends"}), (v_1, v_2, \texttt{"vector spaces"}), (n_1, n_2, \texttt{"numeric fields"}), (m_1, m_2, \texttt{"annotations"})]$$
$$OrderedField\ n_1, Metric\ v_1, Semigroup\ m_1) \Rightarrow d_1 \to d_2 \to d_1$$

Figure 5: Annotated type signature for *atop* with hints, second version

```
type family ShapeHints (xs :: [(*, *, Symbol)]) :: Constraint where
  ShapeHints xs = SH' 1 xs
type family SH' (n :: Nat) (xs :: [(*, *, Symbol)]) :: Constraint where
  SH' n []          = ()
  SH' n ((x1, x2, e) : xs) = (SH' (n + 1) xs, x1 ~ x2 `InconsistentHint` (Text "arg #" :<>: ShowType n :<>: Text " must be a " :<>: Text e))
type family ArgsHints (n :: Symbol) (xs :: [(*, *, Symbol)]) :: Constraint where
  ArgsHints n []          = ()
  ArgsHints n ((x1, x2, e) : xs) = (ArgsHints n xs, x1 ~ x2 `InconsistentHint` (Text n :<>: Text " use different " :<>: Text e))
```

Figure 6: Common type error patterns using type families

and Heeren 2009; Heeren et al. 2003a]. The challenge is to integrate custom ordering with the GHC solver, without losing the facilities for reuse and abstraction dicussed in § 3.

Our solution is to introduce a new combinator:

$$IfNot\ (c :: Constraint)\ (fail :: Constraint)\ (ok :: Constraint)$$

For each constraint c wrapped in this combinator, we specify both how to continue when the constraint is inconsistent – the *fail* branch – and how to continue when that is not known to be the case – the *ok* branch. This is an important point: we also take the *ok* branch if at that point we do not know yet whether the constraint is consistent or not – for example, an equality $\alpha \sim \beta$ may turn out to be inconsistent or not depending on the values we later find for both type variables. By nesting applications of *IfNot* we dictate the order in which constraints are checked. For example, Figure 7 gives part of the type signature of *atop* where constraints are solved in sequentially from top to bottom. As we shall see later, the semantics of *IfNot* ensures that $b_1 \sim b_2$ is only considered by the solver if both $d_1 \sim QDiagram\ b_1\ v_1\ n_1\ m_1$ and $d_2 \sim QDiagram\ b_2\ v_2\ n_2\ m_2$ are consistent with the typing. Knowing this, we can safely point to the difference in back-ends as cause for the error.

Another difference between *IfNot* and the hints from § 3 is that in the former case we can use *TypeError* to signal a problem. The fact that we can use any constraint as *fail* argument to *IfNot* gives us additional flexibility which shall be important to implement some of the type error patterns in § 5. The only restriction is that this *fail* argument must ultimately result in an inconsistency for the whole system to be sound – we discuss how to statically enforce this invariant in Appendix A.

There is one caveat. *IfNot* imposes an ordering between constraints gathered from the same type signature. A custom ordering is also needed between constraints coming from different expressions. For example, when type checking $atop\ d_1\ d_2$, the constraints inside the *IfNot* should only come into play after all the constraints from both d_1 and d_2 have already been considered. Otherwise we run the risk that constraints from *atop* infect those from d_1 and d_2, as was the case in the first example in this section.

Techniques to describe such relations include specialized type rules [Heeren et al. 2003a; Serrano and Hage 2016b] and priorities for those systems based on Constraint Handling Rules [Koninck et al. 2007] such as Chameleon [Stuckey et al. 2006]. Modifying the entire gathering process to accommodate these constructs in GHC is undesirable, since it means changing a big chunk of important code in the compiler. Fortunately, we do not really need complete control over the ordering. The most common use case is to prioritize constraints coming from the arguments in an application node over those coming from the function application itself. In our implementation, you can request this treatment with a pragma:

```
{-# CHECK_ARGS_BEFORE_FN atop #-}
```

Now, whenever $atop\ d_1\ d_2$ is type checked, we are sure that the maximal amount of information has been obtained from the arguments before considering the application itself.

4.1 Implementation

The implementation affects both the gathering phase, which must be made aware of the `CHECK_ARGS_BEFORE_FN` pragma, and the solving phase, which has to take constraint priorities into account and implement the rewriting of *IfNot* constraints.

The modified syntax of constraints now includes optional priorities of the form $Q\ @\ n$. These priorities are generated by the new judgment given in Figure 8. The main change is the addition of a new numeric input n, which syntactically appears after the environment Γ, and which represents the *current generation priority*. This priority is used when creating new constraints, and updated when traversing an application if the head was annotated with the corresponding pragma. As part of this modification, we moved from a rule for application with only one argument to multi-application. This is consistent with how GHC treats applications.

Clearly, the constraint solver must be made to obey these priorities. This entails two different things. First, it should only perform a rewriting step involving a constraint with priority n if no rewriting step can be taken with a constraint with priority larger than n. Note that it does not mean that all constraints with priority m disappear before considering any constraint at $m - 1$: high priority

$$
\begin{array}{lll}
\mathit{atop} :: \mathit{IfNot}\ (d_1 \sim \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1) & (\mathit{TypeError}\ \texttt{"Arg. \#1 to 'atop' must be a diagram"})\ (& \\
\qquad \mathit{IfNot}\ (d_2 \sim \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2) & (\mathit{TypeError}\ \texttt{"Arg. \#2 to 'atop' must be a diagram"})\ (& \\
\qquad \mathit{IfNot}\ (b_1 \sim b_2) & (\mathit{TypeError}\ \texttt{"Back-ends do not coincide"})\ (& \\
\qquad \mathit{IfNot}\ (v_1 \sim v_2) & (\mathit{TypeError}\ \texttt{"Vector spaces do not coincide"})\ (\ldots)))) \Rightarrow d_1 \rightarrow d_2 \rightarrow d_1 &
\end{array}
$$

Figure 7: Annotated type signature for *atop* using *IfNot*

$$
\frac{\overline{\alpha}\ \text{fresh} \qquad (x : \forall \overline{a}. Q \Rightarrow \tau) \in \Gamma}{\Gamma\ ;\ n \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q\ @\ n}
$$

$$
\frac{\alpha\ \text{fresh} \qquad \Gamma, (x : \alpha)\ ;\ n \vdash e : \tau \rightsquigarrow C}{\Gamma\ ;\ n \vdash (\lambda x \rightarrow e) : \alpha \rightarrow \tau \rightsquigarrow C}
$$

$$
\frac{\begin{array}{c}\Gamma\ ;\ n \vdash f : \tau_0 \rightsquigarrow C_0 \qquad \alpha\ \text{fresh} \qquad \Gamma\ ;\ n' \vdash e_i : \tau_i \rightsquigarrow C_i \\ n' = \begin{cases} n+1 & \text{if } f \text{ has a } \texttt{CHECK\_ARGS\_BEFORE\_FN} \\ n & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash f\ e_1\ \ldots\ e_n : \alpha \\ \quad \rightsquigarrow C_0, C_1, \ldots, C_n, \tau_0 \sim (\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \alpha)\ @\ n\end{array}}
$$

Figure 8: Constraint generation with priorities

$$
\begin{array}{lll}
Q & \vdash & \langle \varphi\ ;\ \mathit{IfNot}\ c\ f\ o\ @\ n, Q_a\ ;\ Q_w \rangle \\
 & \hookrightarrow & \langle \varphi\ ;\ c\ @\ n, o\ @\ n, Q_a\ ;\ Q_w \rangle \\
Q & \vdash & \langle \varphi\ ;\ Q_a\ ;\ \mathit{IfNot}\ c\ f\ o\ @\ n, Q_w \rangle \\
 & \hookrightarrow & \begin{cases} \langle \varphi\ ;\ Q_a\ ;\ f\ @\ n, Q_w \rangle & \text{if } c \text{ is inconsistent} \\ \langle \varphi\ ;\ Q_a\ ;\ c\ @\ n, o\ @\ n, Q_w \rangle & \text{otherwise} \end{cases}
\end{array}
$$

Figure 9: Rewriting steps for *IfNot* constraints

constraints, like $Eq\ [\alpha]$, might get stuck, meaning that rewriting can only proceed after additional information has been found, like an assignment to α. When a priority gets stuck it is possible for constraints with lower priority to be considered.

Furthermore, whenever two constraints of the same priority can be rewritten, the one that is not an *IfNot* should take precedence. This design choice ensures that the inconsistency check can take into consideration as much information as possible. For example, if we have $\alpha \overset{1}{\sim} Bool$ and $\alpha \overset{2}{\sim} Int$, with the same priority, but only the second with a custom message, we prefer the compiler to consider first the equality with *Bool*, so that the error is found in $Bool \overset{2}{\sim} Int$ and the custom error message reported.

Second, the constraints generated by a rewriting step inherit the priority of their parent. Interaction between two constraints does not pose a problem: the children are assigned the smallest priority.

The last part of the implementation consists of the custom rewriting of the new constraints, given in Figure 9. In most cases, $\mathit{IfNot}\ c\ f\ o$ is equivalent to the conjunction of c and o, except for the case in which c is inconsistent. This rewriting is sound because the property of f ultimately resulting in an inconsistency is respected. If c is inconsistent and we know it at the moment in which we process the *IfNot*, we can replace it by f. The compiler will not continue further because an error is to be reported. If instead, we only find out that c is inconsistent later, we still have that constraint available because of the second branch of the rewriting.

Detecting inconsistencies while solving. We have already discussed that it is not important that we detect *all* inconsistencies at the moment of rewriting an *IfNot* constraint, since the system remains sound if the inconsistency is found later in the process. But we still have the question of how we detect *any* inconsistency at all: finding this out might require an arbitrary amount of solving.

In our implementation we have decided to only handle the simplest case: type equalities which can be proven to fail even if a later substitution refine the involved types. Type equalities, as discussed in the introduction, are the main source of inconsistencies in a Haskell program. Furthermore, GHC already knows of such a notion: it is called *apartness*, which was introduced alongside closed type families [Eisenberg et al. 2014]. For that reason, the user-facing constraint in our implementation is called:

$$\mathit{WhenApart}\ a\ b\ f\ o \quad \text{representing} \quad \mathit{IfNot}\ (a \sim b)\ f\ o$$

Ensuring soundness. As we have discussed above, the rewriting of $\mathit{IfNot}\ c\ f\ o$ to f whenever c is found to be inconsistent is only sound if f forms an inconsistent set of constraint itself. Otherwise ill-typed code would be able to pass through by wrapping its type signature in an *IfNot* constraint. In order to prevent this scenario, we need to perform an extra check on the type signatures. This check is fully described in Appendix A.

5 TYPE ERROR PATTERNS

The integration of hints as part of GHC's *Constraint* kind opened possibilities for reuse and abstraction in type error diagnosis. Below we discuss a number of scenario's where we can do the same, exploiting the increased power that *IfNot* provides.

5.1 Check in order

The description of *IfNot* constraints has simplicity as one of its advantages: only one new kind of constraint and a new pragma need to be considered by the type checker. However, writing a type signature with nested *IfNot* becomes cumbersome really fast. Our first step is to give DSL authors better syntax: an embedded DSL for describing type error diagnosis for embedded DSLs!

First of all, we need to describe the kind of errors we can rewrite, that is, differentiate between inconsistency and left-undischarged errors. Since, in our implementation, the only inconsistencies we can detect are due to apartness, we bake this fact directly into the data type with the $(:\not\sim:)$ constructor.

$$\textbf{data}\ \mathit{ConstraintFailure} = \forall\ t.t\ :\not\sim:\ t\ |\ \mathit{Undischarged}\ \mathit{Constraint}$$

This data type definition is promoted [Yorgey et al. 2012] by GHC to a kind definition. In fact, we can only use it as a kind since the *Undischarged* constructor needs a *Constraint* as argument. One

remark to make at this point is that the definition of (:≁:) requires types of the same kind: we model homogeneous inequality.

The next step is adding the possibility to attach a custom message to these error conditions. Whether this error message gets converted to *IfNot* or *LeftUndischargedHint* will depend on the error condition itself. In addition, a constraint can be added without any message using *Check*.

As a first approximation, we could define a type signature with custom type errors by merely a list of *CustomError*s. But then once you find the first inconsistency solving completely halts, which means only one type error can be found, even if there are multiple independent ones.

For example, in *atop* we can partition the constraints into three sets so that each set contains mutually independent constraints. The first elements of the partition checks that both arguments are of the shape *QDiagram b v n m*. Each type equality refers to completely different variables, so they have no dependency between them. Once we know that both arguments are diagrams, we move on and check the equality of each pair of type arguments. These checks are again independent of each other. Finally, we check whether types obey the corresponding instances. We prefer to do this at the latest moment to prevent errors coming from under-specified types.

As a result, we use a list of lists instead, each list accounting for one set of independent constraints. The new *CustomErrors* is given in Figure 10. The key point is that when a constraint fails, we still look at the ones in the same partition but not into the next ones.

Using *CustomErrors* in *atop* results in the type signature given in the Introduction in Figure 1. As in the case of hints, we have abstracted non-coincidence of type parameters:

```
type DoNotCoincide what a b
  = a :≁: b :⇛: Text what :◇: Text " do not coincide: "
                :◇: ShowType a :◇: Text " vs. " :◇: ShowType b
```

One difference between this embedding of type errors and the previous one with hints is that an abstraction like *DoNotCoincide* gets assigned kind *CustomError* which is different from *Constraint*.

5.2 Siblings

Another common pattern in custom type error diagnosis is reporting functions which are easily confused with one another, called *siblings* in the literature [Hage and Heeren 2006; Heeren et al. 2003a]. For example, a simplified version of `diagrams` may contain the following two functions to create a color, with and without alpha:

```
rgb  :: Int → Int → Int       → Color
rgba :: Int → Int → Int → Int → Color
```

Now if the user of the DSL writes *rgb* 0 255 0 90, we want the reported error message to point to the fact that *rgb* and *rgba* are often confused. But only when the change is actually a fix. The type signature which we can assign to *rgb* to get this behavior is:

```
rgb :: IfNot     (fn ~ Int → Int → Int       → Color)     -- 1
         (IfNot (fn ~ Int → Int → Int → Int → Color)      -- 2
           (fn ~ Int → Int → Int → Color)                 -- 3
           (TypeError (Text "Try to use 'rgba'")))        -- 4
         () ⇒ fn                                          -- 5
```

In this case, the whole type of the function is wrapped in an apartness check (1). If we find out that the type for *rgb* is compatible with the typing environment, there is nothing left to do. This is the reason why the second branch (5) of the top-level *IfNot* is simply (). If this is not the case, we try to see whether *fn* is compatible with the type of *rgba* instead in (2). If this is the case (4), we then produce the custom type error hinting at the replacement. If we can assign neither type to *fn*, then we prefer to get the original error message, as declared in (3); there is no mention of the use of siblings if the suggestion does not fix the program.

There is one small caveat with this solution, though. Checking whether the type of *fn* is compatible with the one of *rgba* should be done at the very last possible moment. Otherwise, we run the risk of unifying *fn* with *Int* → *Int* → *Int* → *Int* → *Color*, only to find later that this choice was not right. But at this later point in time we have already lost the error message pertaining to *rgba*. If we have more than one sibling, the order in which we check them does not matter, given that the check is done at the end.

We already have discussed a mechanism in the compiler to indicate when a constraint should be considered: priorities. Thus, we just need to introduce a way to declare that a given constraint should be considered as late as possible. In our implementation, such a declaration is done via another type class:

```
class c ⇒ ScheduleAtTheEnd (c :: Constraint)
```

with the special rule that *ScheduleAtTheEnd c* is rewritten to *c*, as the instance declaration indicates, but this later constraint is assigned the lowest priority in the system. In our case we have settled on -1, since gathering may only produce constraints with non-negative priorities. By wrapping our checks for *fn*, we ensure that this choice does not influence further solving, as desired.

The final implementation of the *Sibling* constraint is given in Figure 11. Apart from scheduling the check of the replacement type at the end, the definition also includes an *extra* argument to allow additional constraints to be checked whenever the original type is accepted – the part marked as (5) in the *rgb* type signature.

Our focus in this paper is type error diagnosis for embedded DSLs, but siblings are also useful for general Haskell programming. For example, it is quite common for beginners to forget the difference between these two *Applicative* functions:

```
(⟨$⟩) :: Applicative f ⇒ (a → b) → f a → f b
(⟨$)  :: Applicative f ⇒ a       → f b → f a
```

The solver can be informed now of this fact by means of *Sibling*:

```
(⟨$⟩) :: Sibling "(<$>)" (Applicative f) ((a → b) → f a → f b)
                 "(<$)"                  (a       → f b → f a)
                 fn ⇒ fn
```

Given $f :: Char \to Bool \to Int$, if the user tries $f\ \langle\$\rangle\ [1 :: Int]\ \langle*\rangle$ `"a"` $\langle*\rangle\ [True]$, the resulting error message reads:

```
* Type error in '(<$>)', do you mean '(<$)'
* In the first argument of '(<*>)',
namely 'f <$> [1 :: Int]' ...
```

5.3 Alternatives and conversions

The ideas underlying the *Sibling* construction can also be used to handle another common source of errors in embedded DSLs. In

```
data CustomErrorsPath = Ok | Fail

type family CustomErrors (css :: [[CustomError]]) :: Constraint where
  CustomErrors [ ]        = ()
  CustomErrors (cs : css) = CustomErrors' Ok cs css

type family CustomErrors' p cs r :: Constraint where
  CustomErrors' Ok   [ ]                          r = CustomErrors r
  CustomErrors' Fail [ ]                          r = ()
  CustomErrors' p ((a :∼: b          :⇛: msg) : cs) r = IfNot (a ∼ b) (TypeError msg, CustomErrors' Fail cs [ ]) (CustomErrors' p cs r)
  CustomErrors' p ((Undischarged c :⇛: msg) : cs) r = (c `LeftUndischargedHint` msg, CustomErrors' p cs r)
  CustomErrors' p (Check c                 : cs) r = (c,                             CustomErrors' p cs r)
```

Figure 10: Initial implementation of *CustomErrors*

```
type Sibling nameOk extra tyOk nameWrong tyWrong fn
  = IfNot (fn ∼ tyOk)
      (ScheduleAtTheEnd (IfNot (fn ∼ tyWrong)
        (fn ∼ tyOk)
        (TypeError (Text "Type error in '" :◇: Text nameOk
                   :◇: Text "', do you mean '"
                   :◇: Text nameWrong :◇: Text "'"))))
      extra
```

Figure 11: Implementation of *Sibling*

many of them basic types are wrapped in a custom type in order to differentiate usages and increase type-safety. For example, in `persistent` primary keys are represented by a type *Key e* for a given entity type *e*. Users of the DSL, however, might be tempted to use basic types instead; it can be very helpful then to point them to the right function to perform the conversion.

The `diagrams` library also follows this pattern: points in a two-dimensional space, represented by the type *P2 a*, are distinguished from vectors, represented by *V2 a*. Operations are only defined for those concepts where it makes sense. For example, we can only obtain the perpendicular of a vector, not of a point:

$perp :: Num\ a \Rightarrow V2\ a \to V2\ a$

It is possible that a user tries to compile *perp* $(1, 2)$, since tuple notation is the usual one for vectors in mathematics. The solution in this case is to first call *r2*, which turns the pair into *V2 a*. Let us introduce a new combinator to describe alternatives in the signature of *perp* and suggest the conversion in the right case, as given in Figure 12, including a fallback message. The error message for the expression *perp* $(1, 2)$ is now more useful:

```
* Expecting a 2D vector but got a tuple.
Use 'r2' to turn the tuple into a vector.
```

This does not automatically mean that the stated conversion is the one intended by the user – another fix in this case is writing *perp* (*V2* 1 2). But, as in the case of siblings, the message is only reported if introducing the conversion leads to a well-typed program.

In order to describe this new combinator we need to refine the promoted *CustomError* data type:

```
data CustomError = ConstraintFailure
                 :⇛?: ([CustomErrorAlternative], ErrorMessage)
                 | Check Constraint
data CustomErrorAlternative = Constraint :⇛!: ErrorMessage
```

Now a failed constraint does not immediately turn into an error message, but first checks a list of alternatives, each with a different message associated to it. These alternatives turn into nested applications of *IfNot* via the modified *CustomErrors* type family given in Figure 13. The names $:⇛^?:$ and $:⇛^!:$ are chosen to suggest that whereas in the first case you are still unsure of the type, when you produce the message in the alternatives you know more information about it. The original $:⇛:$ combinator is still available, but now as a synonym for an application of $:⇛^?:$ without alternatives:

```
type cs :⇛: msg = cs :⇛?: ([ ], msg)
```

There is some subtlety related to how these alternatives work that must be taken into consideration by DSL authors. Alternatives should be regarded as different ways to fix the program which are tried in order. But as a result of taking one path, some type variables might be unified. In that respect, they are close to siblings.

In our example with *perp'*, if we take the fail branch of *IfNot* $(v \sim V2\ a)$ it means that we have enough information to distinguish the top-level constructor of *v*, *but nothing else*. If now the first alternative is $v \sim (Int, Int)$ and all we knew about *v* is that it is of the form (a, b), taking this path leads to unifying *a* and *b* with *Int*. This might not be the intended behaviour; in order to ensure that alternatives do not influence other parts of the program they must be of the form $a \sim T\ b_1 \ldots b_n :⇛^!: msg$, for fresh b_1 to b_n.

The examples in this section give convincing reasons that the set of basic combinators in this paper are indeed a good choice. Many common patterns in customizable type error diagnosis are expressible using this language, even some which needed special support inside the compiler in other approaches to the problem.

6 RELATED WORK

There is plenty of work related to improvements for general type error diagnosis. Approaches include graph representations [Hage and Heeren 2006; Zhang et al. 2015], counter-factual typing [Chen and Erwig 2014], type error slicing [Rahli et al. 2015; Stuckey et al. 2006], and interactive type debuggers [Chitil 2001; Stuckey et al. 2003; Tsushima and Asai 2013]. These ideas can be combined with type

```
perp :: CustomErrors [
  [v :∝: V2 a :⇛?: ([v ~ (a, a) :⇛!: Text "Expecting a 2D vector but got a tuple. Use 'r2' to turn a tuple into a vector."]
                  , Text "Expected a 2D vector, but got " :⋄: ShowType v)],
  [Check (Num a)]] ⇒ v → v
```

Figure 12: Annotated type signature for *perp*

```
type family CustomErrors' p cs r :: Constraint where
  ...
  CustomErrors' p ((a :∝: b :⇛?: (alts, msg)) : cs) r
    = IfNot (a ~ b) (ScheduleAtTheEnd (CustomErrorsAlt alts msg (CustomErrors' Fail cs [])))
                    (CustomErrors' p cs r)
  ...
type family CustomErrorsAlt alts fail rest :: Constraint where
  CustomErrorsAlt [] fail rest = (TypeError fail, rest)
  CustomErrorsAlt (((a ~ b) :⇛!: msg) : cs) fail rest = IfNot (a ~ b) (CustomErrorsAlt cs fail rest) (TypeError msg, rest)
```

Figure 13: Extension of *CustomErrors* to support alternatives

error diagnosis for embedded DSLs. In the realm of customizable type error diagnosis we find several approaches:

Post-processing. The support for custom error messages in Idris [Christiansen 2014] is based on reflection over error messages reified as a normal Idris data type. If desired, the default error message can be replaced by a custom message.

An instrumented version of the Scala compiler geared towards type feedback is described in [Plociniczak et al. 2014]. Starting with low-level type checker events, a derivation tree is built on demand. Custom compiler plug-ins are able to inspect this tree and generate domain specific error messages.

The main disadvantage of the post-processing approach is that there is no influence on the actual type inference and checking process. In other words, the compiler still emits the same error messages, which we just "dress up" with domain terms. On the other hand, this leads to a simpler model for error customization, whereas modifying the solving process sometimes leads to surprising results.

Interacting with the type checker. Whereas post-processing takes place in a phase different from type checking itself, other techniques integrate with this part of the compiler, modifying its behavior. We have already mentioned GHC's *TypeError* [Diatchki 2015] and Scala's *@implicitNotFound* [Scala Team 2015] as inspiration for the work presented in this paper.

In the PhD thesis of Wazny [2006] we find a proposal similar to our hints for attaching custom errors to constraints. The main difference is that we use forward propagation: hints are updated as the solving process progresses, whereas in [Wazny 2006] annotations are obtained by backwards reasoning from the offending inconsistency. This difference is mainly due to the way in which each system decides which errors to report: in Chameleon [Stuckey et al. 2006] after an inconsistency is found the system checks for minimal unsatisfiable subsets of constraints to be blamed. In contrast, GHC does not perform such a step, but rather keeps information while solving in order to produce a message.

The Helium Haskell compiler [Heeren et al. 2003b] introduces the notion of *specialized type rules* [Heeren et al. 2003a] as a way to influence the constraints generated by an expression. Constraints might be annotated with custom error messages and/or prioritized during solving [Hage and Heeren 2009]. Later, specialized type rules have been extended [Serrano and Hage 2016b] to consider not only purely syntactical information but also some type information.

Our work is heavily influenced by Helium, but with two main differences. The first is lack of support for matching on different syntactical forms: custom error messages can only be attached to function signatures and type class predicates, and priorities can only be modified between functions and arguments, and with the special *ScheduleAtTheEnd*. The resulting system is therefore less powerful, but still covers many common cases.

The second difference is our deep integration with GHC's type system, in particular with the *Constraint* kind. The specialized type rules of Helium are not part of the Haskell language. In other words, Helium is an external DSL for describing type error diagnosis, while our solution is itself an embedded DSL. As a result, we have all the facitilities of GHC's type-level programming available to us to support abstraction and reuse.

Our work does not consider ways to influence the search procedure of the solver, only the order. But DSLs may have additional invariants, e.g., within a given domain it may be impossible for a type to be an instance of two type classes A and B at the same time. The constraint solver can employ such information and mark the constraint set $A\ t, B\ t$ as inconsistent, in addition to the default inconsistency checking. Known techniques in this area include type class directives [Heeren and Hage 2005], the description of Haskell's type checking as CHR solving [Stuckey et al. 2006] and instance chains [Morris and Jones 2010]. These mechanisms are orthogonal to our work, although interaction between them may give rise to more powerful customization.

Language modifications. Another approach to customization is changing the underlying language itself. The programming environment offered by Racket is well-known for its focus on students [Marceau et al. 2011], offering increasingly complex language levels. Its `syntax-parse` facility also offers a form on errors, but the focus is more on syntactic errors.

7 CONCLUSION AND FUTURE WORK

We presented a set of combinators to describe customizations to the type checking process, including both domain-specific error messages and priorities among constraints. These combinators are integrated in GHC's constraint system, which can be programmed, leading to abstraction facilities which are new in this area, as far as we know. In addition, libraries do not need custom type error diagnosis built-in from the outset. Rather, error diagnosis can be added gradually at a later stage by wrapping the desired functions. This provides DSL authors with a good upgrade path to provide better diagnosis for libraries which are already in use.

Although the focus of the paper is GHC, a specific dialect of the Haskell language, this research shows that constraint reflection, that is, being able to manipulate typing constraints within the language itself, provides a fertile ground for better type error diagnosis. It is interesting to see how much of our work can be translated to other statically-typed languages, especially those already based on constraints [Pottier and Rémy 2005; Swift Team 2016].

There are two main directions of future work. The first one is the interaction of general abstractions, such as *Functor*, *Applicative* or *Monad*, with the use of a specific instance in a piece of code. For example, while writing a parser, programmers often use the *Applicative* operators, but it would be helpful to phrase the error messages using terms related to the domain of parsing. In the current implementation, error messages are attached to type signatures, which live in the general type class.

The second direction is defining new combinators which do not replace default messages with custom ones, but rather serve as an explanation of the solving process. The *Eq* type class in Haskell base libraries provides an example. If *Eq* [*Person*] is left undischarged, the message reports that [*Person*] might be given an instance automatically. However, this is not the best error, since there is already an instance $Eq\ a \Rightarrow Eq\ [a]$. Instead, we prefer to explain that *Eq Person* is needed because of the instance, and that we can give an instance for *Person* if the corresponding **deriving** is added to the data type declaration.

REFERENCES

Max Bolingbroke. 2011. Constraint Kinds for GHC. (2011). http://blog.omega-prime.co.uk/?p=127 Blog post.

Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, 583–594.

Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors *(ICFP '01)*. ACM, 193–204.

David Raymond Christiansen. 2014. Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages. *Presented at TFP 2014* (2014).

Iavor Diatchki. 2015. Custom Type Errors. (2015). Available at https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations *(POPL '14)*. ACM, 671–683.

Jurriaan Hage. 2014. *DOMain Specific Type Error Diagnosis (DOMSTED).* Technical Report UU-CS-2014-019. Department of Information and Computing Sciences, Utrecht University.

Jurriaan Hage and Bastiaan Heeren. 2006. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages, 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers (Lecture Notes in Computer Science)*, Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.), Vol. 4449. 199–216.

Jurriaan Hage and Bastiaan Heeren. 2009. Strategies for Solving Constraints in Type and Effect Systems. *Electron. Notes Theor. Comput. Sci.* 236 (April 2009), 163–183.

Bastiaan Heeren and Jurriaan Hage. 2005. Type Class Directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages (PADL'05)*. 253–267.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003a. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, 3–13.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003b. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, 62–71.

Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996).

Leslie Koninck, Tom Schrijvers, and Bart Demoen. 2007. User-definable Rule Priorities for CHR *(PPDP '07)*. ACM, 25–36.

Hendrik Vincent Koops, José Pedro Magalhães, and W. Bas De Haas. 2013. A Functional Approach to Automatic Melody Harmonisation. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*. ACM, 47–58.

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, 3–18.

Simon Marlow et al. 2010. Haskell 2010 Language Report. (2010). https://www.haskell.org/onlinereport/haskell2010/.

J. Garrett Morris and Mark P. Jones. 2010. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 375–386.

Hubert Plociniczak, Heather Miller, and Martin Odersky. 2014. Improving Human-Compiler Interaction Through Customizable Type Feedback.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electr. Notes Theor. Comp. Sci.* 312 (2015), 197–213.

Scala Team. 2015. Docs for `scala.annotation.implicitNotFound`. (2015).

Alejandro Serrano and Jurriaan Hage. 2016a. *Context-Dependent Type Error Diagnosis for Functional Languages.* Technical Report UU-CS-2016-011. Department of Information and Computing Sciences, Utrecht University.

Alejandro Serrano and Jurriaan Hage. 2016b. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems, ESOP 2016*. 672–698.

Alejandro Serrano and Jurriaan Hage. 2017. Lightweight Soundness for Towers of Language Extensions. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017)*. ACM, New York, NY, USA, 23–34.

Michael Snoyman. 2012. *Developing Web Applications with Haskell and Yesod.* O'Reilly Media, Inc.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive Type Debugging in Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, 72–83.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Lecture Notes in Computer Science, Vol. 4279. 1–25.

Swift Team. 2016. Type Checker Design and Implementation. (2016). https://github.com/apple/swift/blob/master/docs/TypeChecker.rst

Kanae Tsushima and Kenichi Asai. 2013. *An Embedded Type Debugger.* 190–206.

Markus Voelter. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.*

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (2011), 333–412.

Jeremy Wazny. 2006. *Type inference and type error diagnosis for Hindley/Milner with extensions.* Ph.D. Dissertation. University of Melbourne, Australia.

Brent A. Yorgey. 2012. Monoids: Theme and Variations. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, 105–116.

Brent A. Yorgey. 2016. User manual for `diagrams`. (2016).

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion *(TLDI '12)*. ACM, 53–66.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class *(PLDI 2015)*. ACM, 12–21.

A SOUNDNESS CHECK

The check is described as a pair of judgments in Figure 14. Given a type signature $\forall \overline{a}.Q \Rightarrow \tau$, we need to check whether $\boldsymbol{Q} \vdash \epsilon \Vdash Q$ ok. The rules of this judgment traverse the entire tree of constraints, keeping a log of the constraints which have already been visited and are now part of the environment Q. The only non-trivial rule is the one for *IfNot* $c\ f\ o$. In that case, we need to switch to the $\boldsymbol{Q} \vdash Q \Vdash f$ inconsistent judgment for the second argument, whose goal is to check that f is guaranteed to lead to an inconsistency. Primitive constraints are given to the constraint rewriting engine, whose operation must finish with a $\bot$ in the set of residual constraints. A set of constraints is inconsistent whenever at least one of them is inconsistent; in the case of an *IfNot* $c\ f\ o$ constraint both branches f and o must lead to an inconsistency.

In the check we assume that all type families have been expanded. This is enough for most use cases, but breaks when type families use recursion. In that case we need to do fixpoint iteration over the call graph, taking into consideration that under a given environment more than one branch of the type family could be taken.

B THE PATH LIBRARY, ANNOTATED

The path library provides operations to manipulate file paths in a safer way than the usual string-based manipulation. The extra type safety comes from using a specialized data type *Path*, which is parametrized by the base location of the path, which can be absolute or relative, and the type of path it represents, either a directory or a file. For example, you may only extract the file name of a path representing a file,

$$\mathit{filename} :: \mathit{Path}\ b\ \mathit{File} \to \mathit{Path}\ \mathit{Rel}\ \mathit{File}$$

and may only concatenate two paths if the first one is a directory and the second one is relative:

$$(</>) :: \mathit{Path}\ b\ \mathit{Dir} \to \mathit{Path}\ \mathit{Rel}\ t \to \mathit{Path}\ b\ t$$

In this section we define type signatures introducing domain-specific terms in all the basic operations of the path library. This serves as an example of how an existing library can be wrapped, and also as a real world usage of our techniques.

Almost every operation imposes some requirement on either the base or the type of path, so we introduce several synonyms in Figure 15. We do so for all possible combinations of base location and type. In addition, some operations expect the bases of two paths to coincide, which we check by means of *EqualBase*.

Another common mistake in using the library is giving a string directly as an argument, without prior conversion to *Path*. Following the idea of conversions and alternatives introduced in the paper, we can define custom error diagnosis in *CheckIsPath* suggesting parsing the string to get a well-typed program.

Finally we can wrap the operations from the *Path* module. The result is given in Figure 16. The original type signatures are given as comments below the error-annotated one.

B.1 Examples of error messages

The following session of the GHC interpreter shows our customized error diagnosis in use. The first example involves using a string as a path without conversion:

```
> filename "type-errors/Example.hs"
<interactive>:1:1: error:
* You need an explicit conversion from string to Path
using one of the parse* functions.
* In the expression: filename "type-errors/Example.hs"
```

Let us define four (non-existing) files with all combinations of base and type:

```
> let pRelDir :: Path Rel Dir = undefined
> let pAbsDir :: Path Abs Dir = undefined
> let pRelFile :: Path Rel File = undefined
> let pAbsFile :: Path Abs File = undefined
```

We can check that *filename* and *dirname* only accept the corresponding types:

```
> filename pRelDir
<interactive>:2:1: error:
* The given path type 'Dir' does not represent a file.
* In the expression: filename pRelDir

> dirname pAbsFile
<interactive>:3:1: error:
* The given path type 'File' does not represent a dir.
* In the expression: dirname pAbsFile
```

or that *isParentOf* requires bases to coincide and gives a suggestion if given a string:

```
> isParentOf pRelDir pAbsFile
<interactive>:4:1: error:
* The path bases 'Rel' and 'Abs' should coincide:
they have to be either both relative or both absolute.
* In the expression: isParentOf pRelDir pAbsFile

> isParentOf "type-errors" pAbsFile
<interactive>:5:1: error:
* You need a explicit conversion from string to Path
using one of the parse* functions.
* In the expression: isParentOf "type-errors" pAbsFile
```

Finally, we can check how the layered implementation of *CustomErrors* gives back two error messages when the arguments to (</>) do not have the right types:

```
> pAbsFile </> pAbsDir
<interactive>:6:1: error:
* The given path type 'File' does not represent a dir.
* In the expression: pAbsFile </> pAbsDir
<interactive>:6:1: error:
* The given path base 'Abs' is not relative.
* In the expression: pAbsFile </> pAbsDir
```

$$\frac{\mathcal{Q} \vdash \langle id\ ;\ \epsilon\ ;\ Q, P\rangle \hookrightarrow \langle \varphi\ ;\ \epsilon\ ;\ Q'\rangle \qquad \bot \in Q'}{\mathcal{Q} \vdash Q \Vdash P \text{ inconsistent}}$$

$$\frac{\mathcal{Q} \vdash Q \Vdash f \text{ inconsistent} \qquad \mathcal{Q} \vdash Q, c \Vdash o \text{ inconsistent}}{\mathcal{Q} \vdash Q \Vdash \mathit{IfNot}\ c\ f\ o \text{ inconsistent}}$$

$$\frac{\mathcal{Q} \vdash Q, Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n \Vdash Q_i \text{ inconsistent for some } i}{\mathcal{Q} \vdash Q \Vdash Q_1, \ldots, Q_n \text{ inconsistent}}$$

$$\frac{}{\mathcal{Q} \vdash Q \Vdash P \text{ ok}}$$

$$\frac{\mathcal{Q} \vdash Q \Vdash f \text{ inconsistent} \qquad \mathcal{Q} \vdash Q, c \Vdash o \text{ ok}}{\mathcal{Q} \vdash Q \Vdash \mathit{IfNot}\ c\ f\ o \text{ ok}}$$

$$\frac{\begin{array}{c}\mathcal{Q} \vdash Q, Q_2, \ldots, Q_n \Vdash Q_1 \text{ ok} \\ \vdots \\ \mathcal{Q} \vdash Q, Q_1, \ldots, Q_{n-1} \Vdash Q_n \text{ ok}\end{array}}{\mathcal{Q} \vdash Q \Vdash Q_1, \ldots, Q_n \text{ ok}}$$

Figure 14: Judgments checking soundness of *IfNot* constraints

```
type CheckRel  x = x :~: Rel
  :⇛: Text "The given path base '" :◇: ShowType x :◇: Text "' is not relative."
type CheckAbs x = x :~: Abs
  :⇛: Text "The given path base '" :◇: ShowType x :◇: Text "' is not absolute."
type CheckDir  x = x :~: Dir
  :⇛: Text "The given path type '" :◇: ShowType x :◇: Text "' does not represent a dir."
type CheckFile  x = x :~: File
  :⇛: Text "The given path type '" :◇: ShowType x :◇: Text "' does not represent a file."
type EqualBase x y = x :~: y
  :⇛: Text "The bases '" :◇: ShowType x :◇: Text "' and '" :◇: ShowType y :◇: Text "' should coincide:"
      :$$: Text "they have to be either both relative or both absolute."
type CheckIsPath x b t
  = x :~: Path b t :⇛?: ([x ~ String :⇛!: Text "You need a explicit conversion from string to Path"
                                      :$$: Text "using one of the parse* functions."],
                         Text "The given expression is not a path.")
```

Figure 15: Synonyms for shared error messages

```
import Path hiding ((</>), stripDir, isParentOf, parent, dirname, filename, fileExtension, setFileExtension)
import qualified Path

(</>) :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                       [CheckDir t1, CheckRel b2],
                       [Check (r ~ Path b1 t2)]]
      => p1 -> p2 -> r
-- (</>) :: Path b Dir -> Path Rel t -> Path b t
(</>) = (Path. </>)

stripDir :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                          [CheckDir t1, EqualBase b1 b2],
                          [Check (r ~ m (Path Rel t2)), Check (MonadThrow m)]]
         => p1 -> p2 -> r
-- stripDir :: MonadThrow m => Path b Dir -> Path b t -> m (Path Rel t)
stripDir = Path.stripDir

isParentOf :: CustomErrors [[CheckIsPath p1 b1 t1, CheckIsPath p2 b2 t2],
                            [CheckDir t1, EqualBase b1 b2]]
           => p1 -> p2 -> Bool
-- isParentOf :: Path b Dir -> Path b t -> Bool
isParentOf = Path.isParentOf

parent :: CustomErrors [[CheckIsPath p b t], [CheckAbs b], [Check (r ~ Path Abs Dir)]]
       => p -> r
-- parent :: Path Abs t -> Path Abs Dir
parent = Path.parent

dirname :: CustomErrors [[CheckIsPath p b t], [CheckDir t], [Check (r ~ Path Rel Dir)]]
        => p -> r
-- dirname :: Path b Dir -> Path Rel Dir
dirname = Path.dirname

filename :: CustomErrors [[CheckIsPath p b t], [CheckFile t], [Check (r ~ Path Rel File)]]
         => p -> r
-- filename :: Path b File -> Path Rel File
filename = Path.filename

fileExtension :: CustomErrors [[CheckIsPath p b t], [CheckFile t]])
              => p -> String
-- fileExtension :: Path b File -> String
fileExtension = Path.fileExtension

setFileExtension :: CustomErrors [[CheckIsPath p b t], [CheckFile t],
                                  [Check (r ~ m (Path b File)), Check (MonadThrow m)]]
                 => String -> p -> r
-- setFileExtension :: MonadThrow m => String -> Path b File -> m (Path b File)
setFileExtension = Path.setFileExtension
```

Figure 16: Annotated signatures for the path library