Research Article Open Access

Alejandro Serrano* and Jurriaan Hage

A compiler architecture for domain-specific type error diagnosis

https://doi.org/10.1515/comp-2019-0002
Received May 28, 2018; accepted December 17, 2018

Abstract: Domain-specific languages (DSLs) permeate current programming practices. An important kind of DSLs includes those developed and integrated within a host language, which we call embedded or internal DSLs. Unfortunately, embedded DSLs usually fall short on domain-specific error diagnosis, that is, they do not give control to DSL authors over how errors are reported to the programmer. As a consequence, implementation details of the DSL leak through in error messages, and programmers need to understand the internals of the DSL implementation to fix their code in a productive way.
This paper addresses the challenge of building a compiler with integrated support for domain-specific error diagnosis. We assume that the type system is described using a constraint-based approach, and constraint solving is specified using rewrite rules. Domain information can then be injected at constraint gathering time via type rules, during constraint solving via specialized rules and axioms, and finally at blaming and reparation time via transformations. Furthermore, we define error contexts as a way to control the order in which solving and blaming proceeds. We engineer domain-specific error diagnosis in such a way that the compiler can also reuse the techniques for improving general error diagnosis.

Keywords: domain-specific languages, type error diagnosis, constraint-based type inference

1 Domain-Specific languages, domain-Specific errors

Domain-specific languages, or DSLs for short, target a specific domain, as opposed to general-purpose languages. In this way experts in the domain, even without prior programming knowledge, can use the language effectively. Examples abound, like SQL for database processing or HTML for document representation.

There are two approaches to DSL development. One possibility is to make the compiler of the new language a separate entity. Taking this *external* approach implies developing a lot of tooling, including parsers, code generators and static analyzers. This heavy load can be alleviated by the use of frameworks or *language workbenches* [1].

The dual approach is to *embed* the DSL in a general-purpose host language [2], with the aim of reusing most of the already implemented machinery and to ease the integration of several DSLs in the same codebase. From the point of view of the compiler, the programmer is merely using a library. But from our point of view, embedded DSLs are not merely a long list of functions and data types, but typically exhibit common usage patterns. In this paper we focus on DSLs embedded in statically, strongly typed languages such as Haskell [3]. We assume that the reader is familiar with Haskell's syntax, higher-order functions, and type classes.

Alas, the abstraction provided by an embedded DSL is broken when type errors enter the game [4]. Since the compiler has no knowledge of the specific domain, types in error messages are not phrased in domain terms, but may expose details of how the DSL was encoded in the host language. This is a clear disadvantage, because now programmers have to understand not only the DSL terms, but also details of the implementation in order to make sense of the error messages.

For example, the `diagrams`[1] Haskell library includes specific documentation about deciphering error messages. The following is an example error message described in that section:

```
Could not deduce (N a0 ~ N a)
from the context ...
```

***Corresponding Author: Alejandro Serrano:** Utrecht University, The Netherlands; E-mail: A.SerranoMena@uu.nl
Jurriaan Hage: Utrecht University, The Netherlands; E-mail: J.Hage@uu.nl

1 Documentation for `diagrams` is available at http://projects.haskell.org/diagrams/.

The name `N` is not very descriptive; in fact, it is an internal type only advanced user should need to know about. Furthermore, the error does not give any intuition on how to solve the problem. Using the techniques presented in this paper, the error message could be reworded to:

```
Cannot fix the numerical field
in an argument to 'width'.
Please write a type annotation
as described in the manual on page ...
```

This error message mentions the context in which the error occurs, uses domain-specific terms (such as “numerical field”), and points to a well-known solution to that kind of problems. As we discuss later, the context in which the code lives provides additional valuable information which the DSL author can use to tailor the message to the usage pattern that is detected.

In this paper we present a compiler architecture which allows DSL authors to customize the error messages shown to their programmers. Throughout this paper we refer to the person (or team) designing the embedded DSL as *DSL author* and to the person (or team) writing code using a DSL as *programmer* or *user*. We first present the basic architecture of a constraint-based type inference engine (Section 2) divided into three phases: constraint gathering, constraint solving and a final blaming-explanation-and-reparation phase. We then refine this basic architecture by enhancing each of the three phases: we describe the improved architecture in Section 3 and discuss each improvement in detail in Sections 4 to 7. A prototype compiler for the Haskell language based on this architecture has been developed. Finally we describe the limitations of our approach (Sections 8 and 9) and discuss related work (Section 10).

Many parts of the architecture also benefit the explanation of type errors in general, not only those related to domain-specific abstractions. The focus of this paper remains on those techniques which benefit DSL authors, although we describe some integration with general techniques in Section 4.2.

1.1 A simple DSL for drawings

As an example of a simple DSL, yet big enough to showcase our approach to domain-specific errors, we introduce a library to define drawings. The interface is heavily influenced by that of the `diagrams` library. We use Haskell syntax in the code fragments, but the examples are easily translated to other statically-typed languages.

The core of the library is the type `Drawing b v`. A value of such a type is a description of a drawing in vector space `v` using primitives from the back-end `b`. For example, `Drawing SVG 2D` is a two-dimensional drawing which can be represented using Scalable Vector Graphics. Drawings are created using a series of *primitives* which, for the sake of conciseness, we only give the type signatures. Some of these can be used for any back-end:

```
square :: Supports2D b => Float -> Drawing b 2D
circle :: Supports2D b => Float -> Drawing b 2D
sphere :: Supports3D b => Float -> Drawing b 3D
```

while others are only applicable to a specific back-end, like:

```
embed :: Drawing Bitmap 2D -> Drawing SVG 2D
```

which embeds a bitmap into a vector drawing. In these type signatures everything to the left of the => arrow is called a *qualifier*. Each qualifier imposes a constraint on the possible instantiations of each type variable, in this case `b`. In particular, `Supports2D b` is a *class instance* constraint. Readers not familiar with type classes in Haskell may think of them as (distant) relatives to interfaces and traits as found in Java or Scala.

Along with primitives, DSLs usually provide *combinators*. For example, we can juxtapose drawings either horizontally or vertically:

```
hcat :: Drawing b v -> Drawing b v -> Drawing b v
vcat :: Drawing b v -> Drawing b v -> Drawing b v
```

Note that we mandate that the drawings being concatenated agree on both the back-end and vector space in which they live. The reader can think of more ways to combine two drawings, like superposition; the type signatures will be similar to that of `hcat`.

Finally, our library provides a way to show a drawing on the screen via a function

```
show :: ToOpenGL b => Drawing b 2D -> IO ()
```

This function does not return any interesting value, which we represent by the unit type `()`. Scala programmers call this type `Unit`. In this use case `()` is similar to `void` in Java, although the differences between them are deep. In Haskell we also need to declare when a function may perform side-effects, like drawing to the screen or communicating via network. We do so by including `IO` as part of the return type of the function. This fact, however, is not important for the contents of the paper. As an extra constraint, we require that the back-end supports conversion to the graphical library OpenGL.

1.2 The three kinds of type errors

This simple API is enough to highlight the three kinds of type errors which occur during the practice of programming. As discussed in the introduction, the default type errors produced by the compiler do not mention any of the domain-specific concepts, like “drawing”, “back-end” or “vector space”.

1.2.1 Inconsistencies

When a piece of code has an inconsistency, it means that stating that it is well-typed would imply a formula inconsistent with the typing rules of the language. If that is the case, the compiler complains loudly about the problem. Throughout the paper we sometimes show the interaction between the programmer and the compiler, usually done through a Read-Eval-Print Loop or REPL. In those cases we prefix programmer code with > and the compiler output with nothing.

The archetypical example of an inconsistency is forcing a variable in a signature to be instantiated with two different types. For example, if the programmer types the ill-typed expression:

```
> hcat square sphere
```

the compiler complains, because the `b` in `hcat` would need to be `2D` according to the first argument, and at the same time `3D` according to the second,

```
Couldn't match expected type '3D'
               with actual type '2D'
```

The compiler treats any inconsistency of this shape in the same way. However, we can provide a bit more help to the programmer by stating what this inconsistency means in terms of the domain. If the second type parameter of `Drawing` does not coincide in a call to `hcat`, we show:

```
Drawings live in different vector spaces:
  '2D' versus '3D'
```

whereas if the conflict is in the first type parameter we show:

```
Drawings have different back-ends:
  'SVG' versus 'Bitmap'
```

1.2.2 Left undischarged

Suppose now that we have a drawing `d` of type `Drawing SVG 2D` and we want to execute `show d`. Alas, nobody has written the code to translate SVG documents into OpenGL primitives. The requirements for `show` are thus not fulfilled; the compiler informs us with the following message:

```
No instance for (ToOpenGL SVG)
 arising from a use of 'show'
```

The character of this type error is different from an inconsistency: the fact that `ToOpenGL SVG` holds is not inconsistent with the rest of the program or the typing rules. Rather, it is a constraint which is *left undischarged*, but needs to be proven in order for the program to compile.

Note that the same constraint might give rise to different kinds of errors in different contexts. For example, `Supports3D SVG` is a reasonable constraint, since there are third-party libraries which add support for three-dimensional scenes for SVG, but it might be left undischarged if the implementation is not yet there. On the other hand, `Supports3D Bitmap` is an inconsistency, because a 3-D scene cannot be converted to a bitmap without further information about camera, lighting and so on.

1.2.3 Ambiguities

A first-time user of the library might want to try showing some simple drawings on the screen, by writing the following code in the interpreter:

```
> show (circle 1)
```

Unfortunately, the result is not a depiction of the zeroes of $x^2 + y^2 - 1$, but an error:

```
No instance for (Supports2D b0)
 arising from an use of 'circle'
 The type variable 'b0' is ambiguous
```

The problem here is in some sense the reverse of an inconsistency: it is not that we have no possible type assignment for `b0`, it is that we have *more than one*. If we make `b0` equal to `Bitmap` the code to be executed is different from the code in the `SVG` instance. Thus, the compiler has a good reason to reject that program and signal `b0` as *ambiguous*.

Note that this error would not have arisen if `show` had a signature `Drawing b v -> IO (Drawing b v)`. In that case, the compiler is able to give a type to the whole `show circle` expression which still mentions the variables `b` and `v`. A particular choice for those variables is deferred to the use site of the expression. But in the original situation `IO ()` does not mention any variable at all, forcing the compiler to choose at that instance.

In this paper we do not consider ambiguity errors. First of all, it is not clear that this kind of errors benefit from domain-specific error reporting. Second, the way in which ambiguity errors are corrected is uniform: add some type annotations to guide the compiler to choose a value for all type variables.

2 Constraints-and-rules-based type inference

Approaches to type checking and inference are roughly divided into two big groups:

– *Direct* approaches traverse the Abstract Syntax Tree (AST) of the code, performing some computation at each step. By the time all nodes have been visited, the algorithm has determined the correctness of the code and the type of each subexpression. The classical $\mathcal{W}$ and $\mathcal{M}$ implementations of the Hindley-Milner type system [5, 6] use a direct approach.
– *Constraint-based* approaches divide their operation into two phases. The first phase also involves a traversal of the AST, but only to *generate* constraints that the assigned types must satisfy. A dedicated *solver* then is called, which checks whether the constraints are consistent. This process is repeated per binding group, that is, per set of function which form a dependency cycle. The type checker in the GHC Haskell compiler [7], the Swift compiler [8], and Elm compiler [9] are prominent examples. Many language extensions have been described using this approach [10, 11].

Direct approaches have a clear disadvantage with respect to error reporting: since they traverse the AST in a fixed manner, their errors are biased [12]. A constraint-based type checker does not impose such a strict ordering [13, 14]. The solver has a more holistic view of the process, and can decide how to proceed and what part of the program to blame based on the entire set of constraints. Given our end goal of improving error diagnosis, we build our compiler using the constraint-based approach.

The main disadvantage of constraint-based approaches lies in their performance: having two phases means that intermediate results need to be computed and stored. We do not have to pay the full price of constraints, though: we can use a faster alternative, and only when an error is found, run the constraint-based type checker. In many cases, we only need to re-run the checking on the current binding group. This way we reach a good balance between speed and quality of error messages.

The components of a constraint-based type checker are depicted in Figure 1. In addition to the gathering and solving phases, we have an extra *blaming-explanation-and-reparation* phase, which runs only when an error is found. The goal of this phase is to decide which part of the program is responsible for an error (blaming), how to phrase it (explanation) and to suggest ways in which the error can be fixed (reparation).

2.1 Constraint gathering

The first phase of the type checker is usually a syntax-directed process which collects constraints by traversing the tree. We represent it using a judgment $\Gamma \vdash e : \tau \rightsquigarrow C$, meaning that expression e under an environment Γ is assigned a type τ whenever the constraints C are satisfied. This presentation is influenced by the OutsideIn(X) framework for modular type inference [7], which forms the basis of the GHC type checker from version 7.

A variation of the λ-calculus which includes parametric polymorphism and qualifiers is given in Figure 2. Elements of the environment are given a type *scheme*, in which variables may appear quantified and qualifiers are allowed. We distinguish syntactically between *rigid variables*, which appear in those schemes and may be instantiated, and *unification variables*, which represent a yet-to-be-known type. The syntax of constraints is left open, but includes at least equality constraints $\tau \sim \rho$. Constraints are combined into *sets*, the empty set of constraints is represented by $\top$. In this case, any set of constraints may appear as qualifier in a type scheme, but in general constraints form a superset of those allowed in qualifiers.

The constraint gathering judgment is given in Figure 3. For the sake of conciseness, we only focus on the expression language; a real language includes data type declarations and bindings:

– The shape of the judgment $\Gamma \vdash e : \tau \rightsquigarrow C$ requires a type as output. However, the environment Γ contains type schemes instead. The var rule saves this gap by instantiating the corresponding type scheme. That is, we introduce fresh unification variables replacing the quantified variables in the type scheme.
– The abs rule deals with functions. Since we do not know the type of the argument, we introduce a fresh unification variable α in the environment, which will be refined by further constraints.
– Finally, in the case of application, the app ensures that the type of the function parameter coincides with that of the argument by issuing an equality constraint

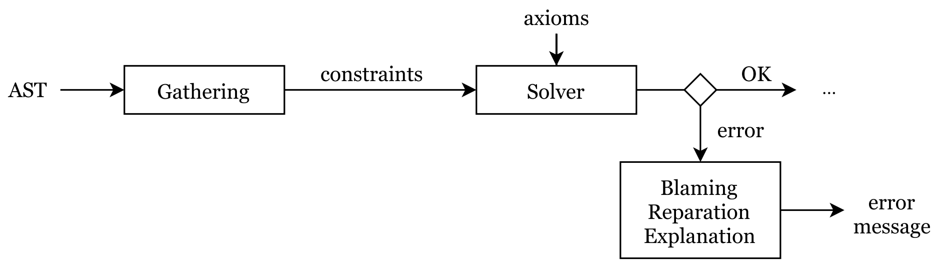

Figure 1: Architecture of a constraints-and-rules-based type checker.

Unif. variables		$\in$	$\alpha, \beta, \dots$
Rigid variables		$\in$	$a, b, \dots$
Type constructors		$\in$	$\mathsf{F}, \mathsf{G}, \dots$
Type vars.	υ, ω	$::=$	$\alpha \mid a$
Monotypes	τ, ρ	$::=$	υ
		$\mid$	$\tau \to \rho$
		$\mid$	$\mathsf{F}\,\tau_1 \dots \tau_n$
Constraints	Q	$::=$	$\tau \sim \rho \mid \dots$
Sets of constraints	C	$::=$	$\top \mid Q \mid C_1, C_2$
Schemes	σ	$::=$	$\forall \overline{a}.\, C \Rightarrow \tau$
Term variables		$\in$	$x, y, f, g, \dots$
Expressions	e	$::=$	$x \mid \lambda x.\, e \mid e_1\, e_2$
Environments	Γ	$::=$	$\epsilon \mid x : \sigma, \Gamma$

Figure 2: Syntax for a variation of λ-calculus.

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

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

$$\frac{\alpha \text{ fresh} \qquad \begin{array}{c}\Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2\end{array}}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow C_1, C_2, \tau_1 \sim \tau_2 \to \alpha}\ \textsc{app}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \rightsquigarrow C \\ \text{There exists a substitution } \theta \\ \text{such that } \theta\tau = \tau^\star \text{ and } \theta C \rightsquigarrow^* C' \subseteq C^\star\end{array}}{\Gamma \vdash e : \forall \overline{a}.\, C^\star \Rightarrow \tau^\star}\ \textsc{top}$$

Figure 3: Constraint gathering for a variation of λ-calculus.

$\tau_1 \sim \tau_2 \to \alpha$. In addition, all the constraints recursively generated by both expressions are conjoined.

When type checking a top-level definition, we assign a type *scheme* to an expression. The rule TOP binds together all the pieces. First, we generate some constraints C. By solving those constraints as discussed in the next section, we obtain a substitution θ and a set of residual constraints C' which cannot be rewritten any further. We need this final set C' to be a subset of the ones stated in the type scheme, that is, the type scheme must provide us with a context strong enough to prove the gathered constraints.

2.2 Constraint solving

Once we know which constraints need to be satisfied, the solver is invoked. In this paper we assume that the solver is a *rewriting* engine: at each step one or more constraints are replaced by other constraints, based on a set of rewriting *rules*. Because of this assumption, we say that our approach to type inference is *constraints-and-rules-based*. Frameworks encompassed by this description include type checking via Constraint Handling Rules [15], Helium [16], OutsideIn(X) [7] and Swift [8].

The rewriting rules dealing with equality constraints for our λ-calculus are given in Figure 4. We use $\top$ to denote an empty set of constraints and $\perp$ to denote inconsistency. This small set of rules already contains rewriting rules of the two possible kinds:

– *Simplification* rules (also known as *canonicalization* rules) have a *single* constraint in the left-hand side. Rewriting a constraint `List Int` $\sim$ `List a` to `Int` $\sim$ `a` is an example of application of one such rule. Intuitively, simplification does not introduce knowledge derived from several parts of the program, it merely turns constraints into a simpler form.
– *Interaction* rules have more than one constraint in the left-hand side. When an interaction rule is applied, information about different parts of the pro-

$$\begin{array}{rcll}
\tau \sim \tau & \rightsquigarrow & \top & \\
\upsilon \sim \mathsf{F}\,\overline{\tau} & \rightsquigarrow & \bot & \text{if } \upsilon \in \mathsf{fv}(\overline{\tau}) \\
\\
\mathsf{F}\,\tau_1\,\ldots\,\tau_n \sim \mathsf{F}\,\rho_1\,\ldots\,\rho_n & \rightsquigarrow & \tau_1 \sim \rho_1, \ldots, \tau_n \sim \rho_n & \\
\mathsf{F}\,\tau_1\,\ldots\,\tau_n \sim \mathsf{F}\,\rho_1\,\ldots\,\rho_m & \rightsquigarrow & \bot & \text{if } n \neq m \\
\mathsf{F}\,\tau_1\,\ldots\,\tau_n \sim \mathsf{G}\,\rho_1\,\ldots\,\rho_m & \rightsquigarrow & \bot & \text{if } \mathsf{F} \not\equiv \mathsf{G} \\
\\
\upsilon \sim \tau_1, \upsilon \sim \tau_2 & \rightsquigarrow & \upsilon \sim \tau_1, \tau_1 \sim \tau_2 & \text{if } \upsilon \notin \mathsf{fv}(\tau_1, \tau_2) \\
\upsilon \sim \tau, \omega \sim \rho & \rightsquigarrow & \upsilon \sim \tau, \omega \sim [\upsilon \mapsto \tau]\rho & \text{if } \upsilon \in \mathsf{fv}(\rho)
\end{array}$$

Figure 4: Constraint solver for equality constraints.

gram, or about dependencies inherent in a single part, is merged. The archetypical example is the substitution of type variable α with τ coming from an assignment $\alpha \sim \tau$ into all other types which mention α.

This distinction is important when explaining why a program is ill-typed. Usually, programmers can easily analyze the implications of simplifying a constraint. Interaction rules are more complicated, since they relate expressions which might be far apart.

2.2.1 Axioms

In order to discharge class instance constraints, the solver uses information from imported modules as an extra source. For example, `Supports2D Bitmap` holds, and thus any constraint of that form can be rewritten to $\top$. Type classes are just one such source: in a language with subtyping the relationships between types also need to be taken into account by the solver. We refer to this extra input as *axioms*, as shown in the architecture in Figure 1.

Following [15], we model axioms as rewriting rules. For example, in Haskell the constraint `Eq (List` τ`)`, stating that the equality function `(==)` is available for lists of type τ, only holds when `Eq` τ is satisfied. We describe the handling of `Eq (List` τ`)` with the following rewriting rule:

$$\texttt{Eq (List } \tau\texttt{)} \rightsquigarrow \texttt{Eq } \tau$$

The general notion of axiom also encompasses those cases in which a certain constraint is just known to hold. We model it by rewriting the constraint to an empty set:

$$\texttt{Supports2D Bitmap} \rightsquigarrow \top \qquad \texttt{Supports2D SVG} \rightsquigarrow \top$$

All the examples of axioms up to this point only involve simplification rules. However, this does not have to be the case. Type families, the Haskell version of type-level functions, give rise to interaction rules [7].

2.3 Blaming, reparation and explanation

Since our solver is a rewriting engine, being finished is the same as being stuck, that is, having no more rewriting steps to apply. The next step is to decide whether the resulting set of constraints implies a valid typing for the corresponding program. At this point, differences between the three kinds of errors emerge:

- When the constraint $\bot$ is present, this means that an inconsistency was found.
- A constraint is left undischarged when it is present in the final set, but we expected to have removed it. For example, when an expression is checked against a type signature in Haskell, all class instance constraints must be discharged by the solver. Otherwise, the caller of the function will not be able to provide all the code needed to execute the function.
- The final possibility is that we expected to find one constraint of a given shape, but we find none. In this case, we have an ambiguity. In our λ-calculus example, for every unification variable υ not quantified in the type signature we must have a constraint $\upsilon \sim \tau$ for some type τ.

As a consequence, the solver is able to detect inconsistencies while it is still running, but constraints left undischarged and ambiguities can only be detected after the solver has finished (otherwise, a rewriting rule may apply and resolve the problem).

Let us consider now the state in which an error has been found. The questions which naturally arise are (1) which parts of the program are *responsible* for the error, (2) how to *communicate* the problem to the programmer, including (3) possible *fixes* to the program. The first problem is known as *blaming*, the second one as *explanation* and the third one as *reparation*. Techniques targeting each of these problems in the context of general programming languages can be found in the literature; the interested reader can find a selection of related work in the PhD thesis of Heeren [14].

The goal of domain-specific error diagnosis is, in fact, to enable these three components to handle domain-specific concepts. It is not enough, though, to modify this last phase of the type checker. In order to obtain good messages, we also have to tweak some aspects of constraint gathering and solving.

To be completely clear, most of the work in error diagnosis consider the problem of obtaining *good* error messages in the *general* case. Our work, in constrast, deals with having *custom* error messages in *specific* scenarios. Both approaches are complementary; in fact we need to ensure that our custom messages do not imply worse error messages in the general case. In fact, most of the methods for good error messages for the general case already use a constraint-based approach to typing, leading to easy integration with our techniques that achieve domain-specificity.

3 A compiler architecture for domain-specific type errors

The basic architecture for a type checker does not differ too much when making it aware of domain-specific concepts. Extra inputs are given in each of the three phases, as Figure 5 shows, in order to customize their behavior. Domain-specific *type rules* (Section 7), *solving rules* and *axioms* (Section 5) and *transformations* (Section 4) are dependent on the DSL the programmer is using, but independent of the program subjected to inspection.

In the original architecture, the constraints and traces of the solving algorithm make up all the information shared between phases. Now constraint gathering produces not only constraints, but decorates them with *error contexts* (Section 6) and *annotations* (Section 4). These pieces of information are queried by both the solver and the blaming-explanation-and-reparation phase. In constrast with the other inputs, error contexts and annotations are generated based on the source code itself.

The end product of the type checker, in the case of failure, is an error message. Messages are usually thought of as monolithic, but we consider them as built from three different parts: trace, reparations and context. The *trace* informs the programmer about the actual problem – an inconsistency or a constraint left undischarged – which led to the error. The trace is a mandatory element of any error message, but domain-specific rules may expand the trace during solving, as we shall see. In some cases, it is possible to discover a *reparation* which fixes the code. If this is the case, the fix is reported to the programmer. For example, our drawing library may have different `rgb` and `rgba` functions depending on whether alpha channel information is given. But these functions can be easily mixed up: if we find that exchanging one for the other fixes the problem, we report it.

```
- Couldn't match expected type 'a -> b'
               with actual type 'Integer'
   The function 'rgb' is applied to too many args
Maybe you wanted to use 'rgba' instead?
```

Traces explain the error itself, but armed with our domain-specific knowledge we can provide a more accurate description. The *error context* embodies this information, like in the following example where it informs the programmer that the problem when merging two drawings lies in their conflicting vector space.

```
Drawings live in different vector spaces:
  '2D' versus '3D'
- Couldn't match expected type '3D'
               with actual type '2D'
```

Error contexts influence not only the error message, but also the precedence when choosing a constraint during solving and blaming.

4 Domain-specific transformations

When an inconsistency is found, or a constraint is left undischarged, we need to find a constraint or set of constraints to blame. Given our constraint-based approach to typing, the blamed constraints will come from the set gathered in the first phase of the type checker. Since each constraint is generated from a specific part of the AST, choosing a constraint is equivalent to choosing a part of the AST (an expression or a declaration).

But what does exactly *being blamed* mean? In most of the literature [17–21], blaming is synonymous to removal:

- If a constraint is left undischarged, we can remove a subset of the original constraints so that the undischarged constraint is no longer generated. That subset of constraints is the one to blame.
- Inconsistency errors are similar, but now our aim is to prevent the error scenario. In order words, we start with an *unsatisfiable* set of constraints and remove constraints until the remaining subset is *satisfiable*.

We find this point of view too restrictive for type error diagnosis. If everything we can do to regain satisfiability of a set of constraints is removing them, there is no way to in-

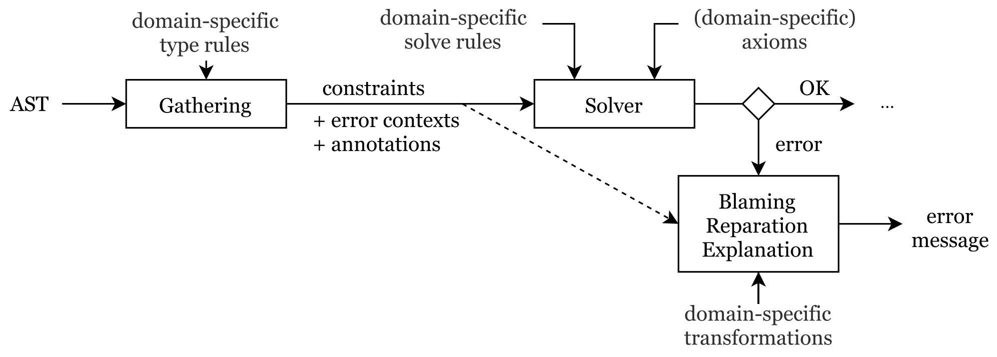
domain-specific type rules
domain-specific solve rules
(domain-specific) axioms
AST
Gathering
constraints
+ error contexts
+ annotations
Solver
OK
...
error
Blaming Reparation Explanation
error message
domain-specific transformations

Figure 5: Architecture of a type checker with domain-specific diagnosis.

ject domain-specific information. The best we can then do is to fine-tune the heuristics the compiler already has for choosing a constraint.

We prefer to see the blaming-explanation-and-reparation phase as a sequence of *transformations* from the original set of constraints into another in which the error is no longer present. Removals are specific cases of transformations, but this general notion encompasses also those described by Hage and Heeren [17] and the isomorphisms found in McAdam's work [22].

Following the spirit of our constraints-and-rules-based approach, we define transformations by rewriting rules which match a set of constraints and transforms them in a predefined fashion. One example of a rewriting rule, based on [22], is to change the type of a function to take one single parameter which is a tuple instead of two separate parameters:

```
fn ~ a -> b -> c  ==>  fn ~ (a, b) -> c
fix "Try to uncurry the function"
```

During the last phase of the type checking and inference process, the compiler applies one or more of these transformations to obtain a new set of constraints. If the transformed set is satisfiable, those constraints from the original set which were subjected to transformation are blamed for the errors. In addition, the messages associated with each of the applied rules, such as `Try to uncurry the function`, are presented as part of the error. The complete sequence is depicted in Figure 6.

A transformation rule which tries to apply uncurrying is an example of a generic rule. But nothing stops the compiler from accepting transformation rules coming from a library, *domain-specific transformations*. The functions `rgb` and `rgba` are well suited for such a rule. In this case, we replace the constraint generated by the former with the one generated if the latter had been found instead:

```
fn ~ Int -> Int -> Int -> Color
  ==>  fn ~ Int -> Int -> Int -> Int -> Color
fix "Extra argument to 'rgb'.
      Did you want to use 'rgba' instead?"
```

That rule is, nevertheless, a bit too general. It applies whenever a function with type `Int -> Int -> Int -> Color` is found. But the fix specifically mentions the `rgb` function. Other pairs of functions, such as `hsv`[2] and `hsva` show the same problem if we only look at the types involved. But if the error involves `hsv`, we definitely do not want to suggest `rgba` as a fix.

This shows the importance of giving transformations information about the source of a particular constraint, in order to decide whether to apply the rule or not. During the gathering process constraints may be given *annotations*. Transformation rules can then query those annotations, and only apply if one of a certain shape is present. For example, we could introduce a `coming-from-rgb` annotation if a constraint is generated from an application of `rgb`:

```
fn ~ Int -> Int -> Int -> Color @ coming-from-rgb
  ==>  fn ~ Int -> Int -> Int -> Int -> Color
fix "Extra argument to 'rgb'.
      Did you want to use 'rgba' instead?"
```

This stops the rule from firing in the case in which `hsv` is used instead of `rgb`.

Annotations are applicable to structures different from specific functions. Consider the case of type checking a conditional. In that case, a constraint c equating the type of both branches is generated. It is impossible to apply a transformation only to c without matching on an annotation, since the constraint has the very general form `tythen ~ tyelse`. The solution is to distinguish the specific scenario by means of a `conditional` annotation:

```
a ~ b @ conditional
```

2 Hue-Saturation-Value, a color model used in computer graphics in addition to Red-Green-Blue (RGB).

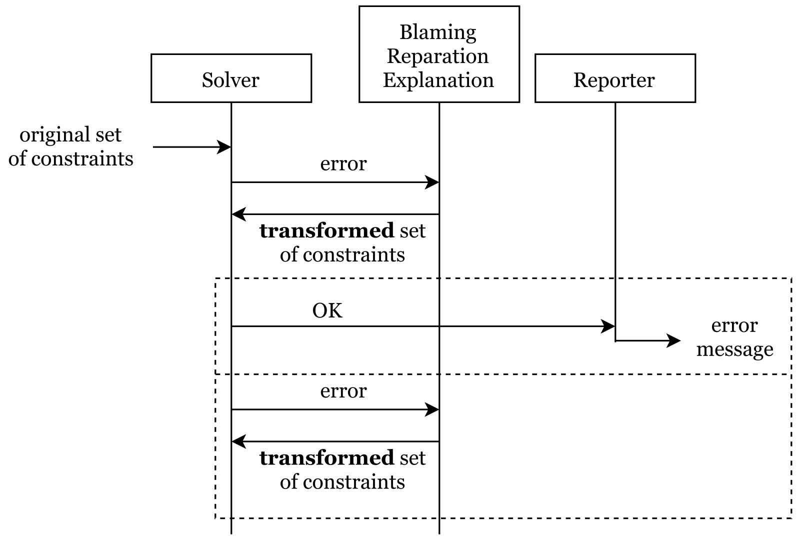
Solver
Blaming
Reparation
Explanation
Reporter
original set of constraints
error
transformed set of constraints
OK
error message
error
transformed set of constraints

Figure 6: Blaming, reparation and explanation using transformations.

```
  ==>  true
fix "Branches of a conditional
      don't have the same type"
```

This rule states that whenever we find an equality coming from a conditional expression, we should try to fix the problem by removing the equality altogether (this is expressed by just replacing it with a true constraint). A similar scenario occurs when type checking an unbounded number of expressions whose types must be compatible, e.g., a list literal. In that case, the reparation process can be instructed to look at all the constraints with the same annotation instead of blaming two specific elements of the list.

4.1 Transformation priorities

Given the same initial set of inconsistent constraints, many different transformations can lead to a satisfiable set. For example, when `rgb` is mistaken for `rgba`, removing the constraint coming from the function altogether instead of replacing it with a new constraint also solves the problem. But it is preferrable to suggest the fix to `rgba` than to merely say that an argument is missing. Thus, we need a way to rank applicable transformations: from all those transformations fixing the problem, the one with the highest priority is then chosen.

As a first approximation, transformations can be ranked *statically*, that is, always prefer rule R_1 over rule R_2. However, as disccused in [17], a more powerful approach is to *dynamically* determine the priority of the rule based on different sources of information. Examples of those sources are in how many different errors a given constraint is involved (since you prefer to suggest one transformation which fixes many problems over several independent transformations), and the place in the AST which generated the constraint (smaller subexpressions might be preferrable to those nearer to the root).

Once again, prioritizing the replacement of `rgb` by `rgba` over another transformation is a decision independent of the program being type checked. But in other cases these priorities depend on the structure of the program, like preferring to blame constraints coming from one specific argument of a function rather than these from the other. In order to do so we overlay a *blaming order* on top of the gathered constraint set. This blaming order is part of the error context structure discussed in Section 6.

4.2 Integration with other general type error diagnosis

The architecture described here allows integration with most of the approaches to general type error diagnosis in the literature. We have already mentioned heuristics [17] and isomorphisms [22], but minimal unsatisfiable subsets [18], Bayesian reasoning [19], MaxSMT solving [20] and correcting sets [21] are other interesting techniques.

The key point is that any approach to type error diagnosis is ultimately a way to search for possible transformations of either the program text or the generated constraint set. Thus, any of these techniques can be applied to an inconsistent constraint set, interpreting the results as an additional transformation rule. The compiler writer has to decide how to rank this transformation with respect to the domain-specific ones: preferring specificity over generality is typically a good approach. Most of these techniques only return one possible set of blamed constraints; if more are produced, all of them have to be ranked.

4.3 Making it practical

The idealized description of blaming as a search over all possible transformations is not practical, though: potentially every rule could match every possible constraint, leading to an exponential amount of work. We can refine our procedure to obtain better performance:

- We do not need to consider the complete original set of constraints as input to the sequence of transformations. We can restrict this input to those constraints involved in the inconsistencies found by the solver, and those involved in traces to left undischarged constraints.
- Since transformation rules are prioritized, we only need to generate lower priority transformations once we have proven that none with a higher priority works. The compiler may also implement policies, like preferring a smaller sequence of transformation over a larger one, in order to prune the search space effectively. There is also room for more complex heuristics, like removing intermediate steps in a long sequence of transformations, in order to ensure that the sequence is minimal.
- In our description so far, the information obtained from the second and subsequent phases of solving is thrown out. However, we can feed this information back to the blaming-explanation-and-reparation phase to guide search. For example, if a certain sequence of transformations leads to a new solution which is still inconsistent as a whole, but where the number of $\perp$ constraints is smaller, the compiler may decide to extend the already known sequence instead of starting anew.

5 Domain-specific solving rules and axioms

In most type checkers, axioms can introduce domain-specific knowledge, since they depend on the specific libraries and modules used by the source code being compiled. Type classes in Haskell are a prime example of this feature. We say that those axioms and rewriting rules which can be expressed using constructs from the host language are *internal*. Special constraints to encode domain-specific error conditions in this internal fashion are gradually appearing in some compilers [24, 42].

There are many DSLs for which internal rules are sufficient to describe how their components ought to be composed, but not for all of them. For that reason, techniques have been developed to directly influence the solving process; the corresponding rules are *external* to the host programming language. Chameleon [25] and GHC type checker plugins [26] are examples of systems with such an external domain-specific constraint solving support.

External solving rules and axioms pose a big challenge to the architecture of most type checkers. In many cases, the result of type checking is used to generate code in further stages of the pipeline – a process usually called *elaboration*. One such example is that of Haskell type classes, which are turned into dictionaries. When providing support for domain-specific solving, a compiler must also provide an interface to the elaborator. The scenario is simpler when external rules are constrained to end in an error state, because there is no elaboration involved.

Our compiler architecture is able to accommodate both internal and external domain-specific solving rules and axioms. The choice depends on the expressive power that the compiler writer wants to put in the hands of DSL authors.

In Section 2.3 we pointed out that inconsistencies during type checking are represented by $\perp$ constraints in the final constraint set. However, merely having a $\perp$ constraint does not convey information useful to the programmer: infinite types, failed unifications and so on, are treated uniformly. The solution is to attach a message to each of these $\perp$ constraints, depending on the rewriting rule leading to it. For example, the solving rules for the variation of λ-calculus given in Figure 4 leading to inconsistencies are extended as shown in Figure 7.

These messages attached to $\perp$ constraints make up the *trace* of the error message, as defined in Section 3. But they are not the only providers of such information: other rewriting rules may also add messages to the trace, which provide valuable information to explain how the solving has proceeded. When an error is generated, all the messages along the way are used to build the trace.

As an example, let us consider a type system featuring a constraint for type instantiation. The following con-

$$\begin{array}{rcl} \upsilon \sim \mathsf{F}\,\overline{\tau} & \rightsquigarrow & \bot \quad \text{if } \upsilon \in \mathsf{fv}(\overline{\tau}) \\ & \textsf{error} & \texttt{Occurs check: cannot construct infinite type } \mathsf{F}\,\overline{\tau} \\ \mathsf{F}\,\tau_1\,\ldots\,\tau_n \sim \mathsf{G}\,\rho_1\,\ldots\,\rho_m & \rightsquigarrow & \bot \quad \text{if } n \neq m \\ & \textsf{error} & \texttt{Cannot unify: different type constructors } \mathsf{F} \texttt{ and } \mathsf{G} \end{array}$$

Figure 7: Extended constraint solver with error information.

straint holds in such a system,

```
    ∀a. Eq a => a -> a -> Bool
      ≤ ∀b. Ord b => [b] -> [b] -> Bool
```

since the `a` in the left-hand side can be unified with `[b]`, and `Ord b` implies `Ord [b]`, which in turn implies `Eq [b]`. In contrast, the constraint

```
    ∀a. Eq a => a -> a -> Bool
      ≤ ∀b. Ord b => [b] -> [b] -> Int
```

does not hold, because the return types are different. The inconsistency is due to `Bool` $\not\sim$ `Int`, but reporting only that fact might be misleading. What we aim is to have:

```
- Couldn't match types 'Int' and 'Bool'
- During the instantiation
    of 'forall a. Eq a => a -> a -> Bool'
    to 'forall b. Ord b => [b] -> [b] -> Int'
```

The second message is obtained from the simplification rule which applies in the case of an instantiation constraint:

$\tau \leq \forall \overline{a}.\, Q \Rightarrow \rho \;\rightsquigarrow\; \ldots$
  trace `During the instantiation of` τ `to` $\forall \overline{a}.\, Q \Rightarrow \rho$

Note that the extra explanation is shown only when one (or more) of the constraints generated by this rule leads to an inconsistency.

Rewriting rules can also add messages to be shown when a constraint is left undischarged. For example, suppose that our drawing library is not a monolithic library, but it is split into several packages, each of them defining support for a different image format. If we are missing an instance for `ToOpenGL`, we can advise the programmer to include an extra library in their project.

`ToOpenGL` τ undischarged
  `I cannot find OpenGL support for` τ
  `Are you missing a drawings-format-* lib?`

In summary, we propose to reify the error trace during solving. This allows us to model how error messages arise, and how to introduce multiple explanation steps. In the presence of domain-specific solving rules and axioms, this reification becomes a rich substrate for custom type error diagnosis.

6 Error contexts

Domain-specific reparations, rules and axioms influence type checking uniformly across programs. The reason is that they operate at the level of constraints, where the connection with the original AST is lost or at least weak. One of the motivations for separating constraint gathering and constraint solving is to make the process independent from the way in which the AST is traversed. In contrast, good domain-specific error reporting heavily depends on contextual information, as discussed in Section 3. The same error from the point of view of the solver should generate different messages for different expressions.

Take for example a disagreement on vector spaces, `2D` $\sim$ `3D`. When the error comes from composing two drawings, as in `hcat d1 d2` we prefer a message unbiased between the drawings `d1` and `d2`:

```
Drawings live in different vector spaces:
  '2D' versus '3D'
```

On the other hand, if `show d` is type checked and `d` is not a two-dimensional drawing, we shall rather blame the drawing than the function `show`:

```
Only 2D drawings can be shown,
but the argument is 3D
```

The inconsistency itself is the same in both cases, `2D` $\sim$ `3D`, but the contexts in which they appear (arguments to `hcat` versus argument to `show`) determine what error message we prefer in each situation.

A first approach to context-dependent error messages is to annotate each constraint with a message [27]. How to deal with these annotations depends on whether they are propagated at solving time (forward propagation) or whether they are just considered once a constraint is blamed (backward propagation). Backward propagation depends heavily on the blaming phase. In a scenario when more than one constraint is annotated with domain-specific information, it is difficult to specify what is the right message to show.

Forward propagation poses problems during solving. Simplification rules are easy to handle: since there is only one constraint in the left-hand side, propagation can just copy it to the new constraint. Alas, when several constraints interact, how to choose between the potentially different error messages from the constraints being rewritten? Just aggregating all of them in a big list leads to huge stacks of messages, which can potentially distract the programmer from fixing the actual error.

As a result, forward propagation needs a *policy* to decide how to merge different error messages when several constraints interact. Such a policy needs to be carefully defined to not throw away any relevant information and at the same time not keeping too much of it around. Furthermore, this policy needs to be open to domain-specific modifications, because the relevance of some details in relation to others depend on the domain.

We have decided not to take this route, because it involves adding one more specification language to the process, namely one to describe policies. Furthermore, our solution using error contexts allows us to tackle another important problem for good type error diagnosis: ordering of constraints during solving and blaming.

6.1 Order matters

Before describing our solution for introducing context dependence in error messages, let us consider another problem DSL authors must face when designing good error reporting for their libraries: *bias* in solving and blaming.

Going back to our drawings, consider a juxtaposition, `hcat d1 d2`, where each drawing is not merely a variable, but a complex expression. As a consequence, the constraint set given to the solver includes constraints C_1 arising from the first drawing, constraints C_2 from the second drawing and some extra constraints $C^{\rightarrow}$ coming from the application. From the point of view of the solver, though, this distinction does not exist, there is just an unordered set of constraints.

Given its non-deterministic nature, the solver may consider constraints in $C^{\rightarrow}$, like `v1 ~ v2` and `b1 ~ b2`, first. No error would be reported at this point, because those variables are still unconstrained. Then, it continues with constraints from C_1, which fix `v1` and `v2`. It is only when C_2 is considered that the error pops up. This specific ordering does not lead to the best possible error message: it is better to consider all constraints from C_1 and C_2 before moving to $C^{\rightarrow}$. Choosing an order imposes a bias.

At first sight it seems plausible to unbias solving, and some algorithms have been devised for specific type systems [12]. However, the picture for general constraint solving is much worse: if you have three constraints c_1, c_2 and c_3 such that each pair is inconsistent (for example, `a ~ Int`, `a ~ Bool` and `a ~ Char`), the first two that you choose to interact imply which inconsistency is found. It may not be possible to try other pairs to find all possible inconsistencies, and it is clearly not performant.

Let us make a virtue of necessity. We cannot fight bias completely, but we can make *bias work to our benefit*. For example, by solving all constraint from C_1 and C_2 before $C^{\rightarrow}$, we ensure that whenever an error is reported for a $C^{\rightarrow}$ constraint it really comes from the dependencies between the arguments to `hcat`.

Bias also plays a role in blaming. Transformations as described in Section 4 come with a ranking, which means that they favour some kind of expressions and errors over others. The general choice may not suit some domain-specific constructs, so we introduce also an ordering which the blaming engine must observe.

We already hinted at one example of blaming precedence at the beginning of this section: when the programmer writes `show d` we prefer to discover the drawing `d` as the culprit of an inconsistency rather than `show`, based on the domain-specific knowledge that programmers forget that some primitives are not allowed in a two-dimensional context more often than they mistake the name of the function `show`. Note that this reasoning does depend on knowledge of the DSL author about other facilities featured in their DSL: in another scenario in which we have both `show2D` and `show3D` functions, biasing blaming towards the argument is not such a great idea.

6.2 The solution

In order to handle these requirements we move from bare constraint sets to an organization of constraints in *error contexts*:

– Each error context holds one or more constraints.
– Error contexts for a given expression form a tree. This structure is used by the solver to give precedence to some interactions over others.
– We optionally associate a message with each context. The message is shown if the blamer decides that the error has occured within that context.
– We can also influence the blaming-explanation-and-reparation phase by giving precedence to one context over another. In contrast to error contexts, in which we require a tree, a directed acyclic graph is enough to define this set of preferences.

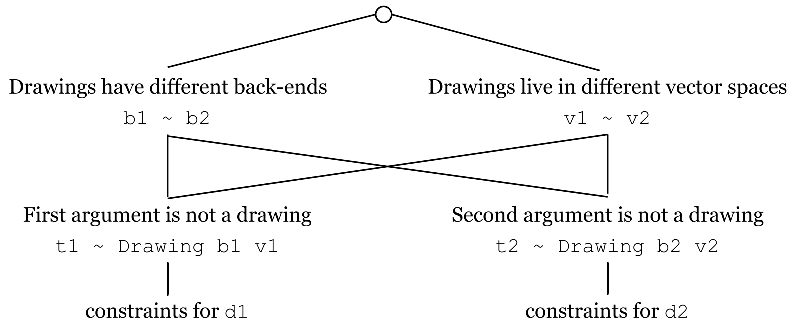
Drawings have different back-ends
b1 ~ b2
Drawings live in different vector spaces
v1 ~ v2
First argument is not a drawing
t1 ~ Drawing b1 v1
Second argument is not a drawing
t2 ~ Drawing b2 v2
constraints for d1
constraints for d2

Figure 8: Error contexts for a call `hcat d1 d2`.

We depict error contexts for the expression `hcat d1 d2` in Figure 8. Intuitively this graph instructs the solver to first work on the constraints gathered from each subexpression `d1` and `d2` independently. Then, the type of each subexpression is matched against `Drawing b v`: the context states that whenever an error happens in this phase, we should explain that the corresponding argument is not a drawing. If no error is found, we move to checking the equality of vector spaces and back-ends, each of the checks happens within a different context.

Formally, error context information for a set of constraints Q from a piece of code is a tuple $\langle \mathcal{E}, \prec_S, \prec_B, \kappa \rangle$. The set of error context identifiers $\mathcal{E}$ forms a partially ordered set with respect to the solving relation $\prec_S$ with a maximum Ξ – this is equivalent to giving a tree structure over $\mathcal{E}$ – and a pre-ordered set with respect to the blaming relation $\prec_B$. The function $\kappa : Q \to \mathcal{E}$ assigns to each initial constraint its *initial context*. Note that the message associated with each context is not really used until the error message is shown: solver and blamer are agnostic to this part of the context information.

For example, the tree depicted in Figure 8 is formalized by introducing five error contexts κ, κ_b, κ_v, κ_1, and κ_2, which correspond to the nodes of the tree when read top-to-bottom, left-to-right. The order represented by the edges is given as:

$$\kappa_b \prec_S \kappa_1, \kappa_b \prec_S \kappa_2, \kappa_v \prec_S \kappa_1, \kappa_v \prec_S \kappa_2,$$
$$\kappa \prec_S \kappa_b, \kappa \prec_S \kappa_v$$

Since $\prec_S$ is a partial order with maximum element, we can define a join operation $\sqcup$ between error contexts such that $e = e_1 \sqcup e_2$ if and only if e is the smallest context such that $e \prec_S e_1$ and $e \prec_S e_2$. We extend this operation to non-empty sets of constraints. Using this operation, we define the *minimum solving error context* of a subset $C' \subset C$ of constraints as:

$$\kappa(C') = \bigsqcup_{Q \in C'} \kappa(Q)$$

Visually, the minimum error context is found by climbing in the context graph until we find the first common ancestor of all the constraints in C'. For example, the minimum solving context of `t1` $\sim$ `Drawing b1 v2` and `v1` $\sim$ `v2` in Figure 8 is given by $\kappa_1 \sqcup \kappa_v = \kappa_v$.

Every constraint considered by the solver, in particular each inconsistency, can be traced back to one such subset C' of the initial set of constraints. The end goal of the order $\prec_S$ is that constraints closer in the context graph interact before constraints that are far apart. We say that an inconsistent set of constraints C' is *context-minimal* if there is no constraint $Q \in C'$ which is part of other inconsistent set $C^\star$ such that $\kappa(C') \prec_S \kappa(C^\star)$.

Our requirement for the solver can now be stated: we want every inconsistency to originate from a context-minimal set of constraints. If this is not the case, then the solver allows constraints to interact too early with respect to the desired order. Our architecture does not impose a specific way in which this requirement must be satisfied, but the interested reader can consult [28] for an implementation using Constraint Handling Rules.

The order $\prec_B$ is introduced as part of the ranking during blaming and reparation. As in the case of solving order, we first need to extend this relation to sets of constraints. However, in this case we do it differently: we say that $C_1 \prec_B C_2$ if every $Q \in C_1$ is greater than or incomparable to every $Q' \in C_2$, and at least one of them is strictly greater (if every pair of constraints is incomparable, then the sets are also incomparable). That is, whereas for solving we consider the context of all constraints, precedence of only one constraint is enough to bias blaming.

Suppose for example that in the scenario given in Figure 8 we prefer to blame the second drawing instead of the first one. This is a reasonable choice: we assume the information from the first argument is correct, and the second one should adhere to it. In that situation we impose $\kappa_2 \prec_B \kappa_1$. Our definition for sets of constraints ensures that if we ever get to blame the first argument, no constraint arising from the second argument is present; otherwise the $\kappa_2 \prec_B \kappa_1$ condition is enough to skew the blaming towards κ_2.

Implementation-wise, we build upon the priorities introduced in Section 4 for the reparation phase. Now, prior to checking the priority from transformations, we check the $\prec_B$ relation between the matched set of constraints. If the outcome is that the sets are incomparable or equal – the most common case – we rank the rewritings as explained in Section 4. In other words, the blaming order as defined by error contexts takes precedence over the general one.

7 Domain-specific type rules

Going back to Figure 5 we see that domain-specific reparation rules and axioms are direct inputs of the corresponding phase. On the other hand, annotations used in the reparation stage and error contexts for solving and blaming superimpose their structure on the constraints in the system. Thus, the gathering phase needs to return not only a constraint set, but also annotations and error contexts. The question is then: how is this data constructed during the gathering phase?

The gathering phase proceeds by traversing the AST. This process is syntax-directed: for each shape of an expression in the language syntax there is one *type rule* which matches, and declares the constraints to be generated. We recall here the type rule for application in our variation of the λ-calculus:

$$\frac{\alpha \text{ fresh} \qquad \begin{array}{l} \Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2 \end{array}}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow C_1, C_2, \tau_1 \sim \tau_2 \to \alpha}\ \text{APP}$$

This rule is no longer sufficient, since it does not indicate the error context in which each constraint lives. Supposing that error contexts are created top-down and constraints are collected bottom-up, as in this case, we extend the judgment to $\Gamma, \kappa \vdash e : \tau \rightsquigarrow \hat{Q}\ ; \mathcal{E}$: the error context κ in the left-hand side represents the current error context, and each new constraint is annotated with its context. In addition, the information about error contexts $\mathcal{E}$ is merged. In the type rules we represent the merge of error context information using $\cup$. Note that this is different from the join of error context themselves, which we represent by $\sqcup$. A simple extension of the application rule which just keeps the same error context as given follows:

$$\frac{\alpha \text{ fresh} \qquad \begin{array}{l} \Gamma, \kappa \vdash e_1 : \tau_1 \rightsquigarrow \hat{C}_1\ ; \mathcal{E}_1 \\ \Gamma, \kappa \vdash e_2 : \tau_2 \rightsquigarrow \hat{C}_2\ ; \mathcal{E}_2 \end{array}}{\Gamma, \kappa \vdash e_1\, e_2 : \alpha \rightsquigarrow \hat{C}_1, \hat{C}_2, (\tau_1 \sim \tau_2 \to \alpha) @ \kappa\ ; \mathcal{E}_1 \cup \mathcal{E}_2}\ \text{APP}$$

Extending all rules in the same fashion (that is, keeping the same context) makes the whole idea of error contexts useless, since there are no preferences being stated.

Domain-specific *type rules* [13, 30] allow DSL authors to modify the gathering process. The idea is to specify a gathering rule for expressions of a specific shape which takes precedence over the default type rules. A rule for `hcat` d_1 d_2 which results in contextual information as described in Figure 8 is given in Figure 9. The rule might look a bit intimidating at first, but let us consider one part at a time:

– In the type rule, d_1 and d_2 are *metavariables* which stand for any possible subexpression. As a consequence, the rule matches any application of `hcat` to two arguments.
– The first line just recursively asks for the constraints and types of d_1 and d_2.
– The next five lines declare four new error contexts, along with the associated messages and how they are ordered. Note that the new contexts κ_1 and κ_2 are the ones set as current when gathering constraints for arguments.
– The last two lines declare two new constraint sets along with error context information. These constraints are added to those from d_1 and d_2 in the consequent.

It is important to realize that we are not attaching the domain-specific type rule to the function `hcat` itself, but rather to expressions of the form `hcat` d_1 d_2. That means that if the function is used with fewer arguments, like using `zipWith hcat ds1 ds2` to construct a list of drawings by putting together drawings from other two lists, the default type rule is still applied. Making type rules match on specific syntactic constructs allows us address the specific patterns in which a particular DSL is used.

This form of type rules is not restricted to matching applications of a function to a number of arguments. Type rules for expressions such as `hcat` d_1 `(vcat` d_2 d_3`)` are also allowed. However, they fall short for a common usage pattern in DSLs where a given combinator is applied several times in a row. For example, using the `Applicative` type class leads to expressions of the form `f <$> e1 <*> ... <*> en`. Or even simpler, constructing a list is done by repeated application of the cons constructor, `e1 : ... : en : []`. A (precarious) solution is to have several copies

$$\Gamma, \kappa_1 \vdash d_1 : \tau_1 \rightsquigarrow \hat{C}_1 \, ; \mathcal{E}_1 \qquad \Gamma, \kappa_2 \vdash d_2 : \tau_2 \rightsquigarrow \hat{C}_2 \, ; \mathcal{E}_2$$

κ_1 new context with message `First argument is not a drawing`
κ_2 new context with message `Second argument is not a drawing`
κ_b new context with message `Drawings have different back-ends`
κ_v new context with message `Drawings live in different vector spaces`

$$\frac{\begin{array}{rcl} \mathcal{E}' & = & \kappa_b \prec_S \kappa_1, \kappa_b \prec_S \kappa_2, \kappa_v \prec_S \kappa_1, \kappa_v \prec_S \kappa_2, \kappa \prec_S \kappa_b, \kappa \prec_S \kappa_v \\ \hat{C}' & = & (\tau_1 \sim \texttt{Drawing}\, b_1\, v_1) \mathbin{@} \kappa_1, (\tau_2 \sim \texttt{Drawing}\, b_2\, v_2) \mathbin{@} \kappa_2 \\ \hat{C}'' & = & (b_1 \sim b_2) \mathbin{@} \kappa_b, (v_1 \sim v_2) \mathbin{@} \kappa_v \end{array}}{\Gamma, \kappa \vdash \texttt{hcat}\ d_1\ d_2 : \texttt{Drawing}\ b_1\, v_1 \rightsquigarrow \hat{C}_1, \hat{C}_2, \hat{C}', \hat{C}'' \, ; \mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}'}$$

Figure 9: Domain-specific type rule for `hcat d1 d2`.

of the rule for different number of appplications of the operator. A possible extension is a facility for domain-specific type rules to match iterative or recursive shapes of expressions, as described in [29].

Let us consider now the scenario in which we want to integrate our drawings library with a common abstraction in Haskell code, namely the `Monoid` type class. Implementing that type class means that programmers can use the familiar <> operator to juxtapose two drawings, the same used to concatenate two strings or combine two optional values. However, at the point in which we use the function <> we do not know which is the intended usage – drawings, strings, or something else – which hinders the possibility of domain-specific type error diagnosis.

A solution to this problem is to run the type checking pipeline one additional time for the last binding group (the one which generated the type error) [30]. When the first run of the solver finds inconsistencies or undischarged constraints, the gathering process is restarted. However, in this second stage it receives as extra input a set of satisfiable constraints. These constraints are obtained by pruning the original constraint set, and represent the type information that we derived from our code. In the second traversal of the AST, some of the type rules may only fire if some information can be defined for the satisfiable subset. For example, a type rule for <> referring to drawings might only apply if it detects that at least one of its arguments is known to be of type `Drawing b v`.

7.1 Soundness and completeness

Domain-specific type rules, if applied carelessly, have the power to subvert the type system. Nothing in the described architecture stops the domain-specific rule for `hcat` from stating that the result type is `Bool`. Clearly this is undesirable, because such a rule will cause the compiler to malfunction.

Unsoundness is not the only problem which can affect type rules, they can also be overly specific. We described our `circle` primitive as having the type

```
circle :: Supports2D b => Float -> Drawing b 2D
```

If a domain-specific type rule says that now `circle :: Float -> Drawing SVG 2D`, the compiler does not break. But some programs which used to compile may stop from doing so, because the type is now less general. Whether this is a good thing depends on the context in which the DSL is used. For example, restricting the type of some combinators in a “beginner” version of a DSL usually leads to better error messages.

In order to prevent these scenarios, a compiler featuring domain-specific type rules should check for the *soundness*, and optionally *completeness*, of those rules prior to applying them. These checks are done by comparing the constraint sets resulting from the new type rule against the default type rules [13, 30]. When type rules become more complicated, like those featuring recursion, it is useful to consider them as language extensions and apply the techniques from that area [31, 32].

8 Limitations

The architecture presented in this paper is applicable to compilers whose type checker has a constraints-and-rules-based approach. The approach has some limitations which we plan to address in future work.

8.1 Backtracking

An implicit assumption during the description of the type checker is that the rewriting rules of the host language do not allow backtracking. In other words, once a rule is applied to a set of constraints, there is no way to get back to the original constraints to try other possible rules. This assumption holds for most programming languages, although there are some exceptions like Swift [8], which uses backtracking to support overloading of operators such as +.

Backtracking opens up a can of worms. First of all, if an inconsistency is detected, this implies that every possible choice of rule applications led to an error. But these errors might be different and involve different parts of the program. Thus, the type error reporting mechanism needs to work much harder to explain all of these possibilities. Another issue that arises is to decide which constraints are blamed and selected for reparation.

In the field of domain-specific error diagnosis, backtracking adds a new point of choice: in which order branches are explored. The extensions to Constraint Handling Rules described in [33] provide a first step in that direction. However, we foresee problems in scenarios where multiple rules may give rise to multiple choice points, and thus to different domain-specific diagnoses.

8.2 Binding in type rules

In our description of type rules, we have only considered examples involving function application. This is not a restriction of our approach: the compiler writer decides which parts of the host language are open to be overridden.

Those constructs involving binding, such as abstractions or local declarations, introduce additional problems. First of all, we cannot use simple syntactic unification, because α-equivalent terms should be considered equivalent when matching. We also have to handle name capturing in the right way to preserve the behavior of the code. Finally, soundnesss and completeness checks become more involved, since they need to handle name introduction.

Having said so, our experience is that most DSLs would not benefit from this extra power, since binding is usually thought of as part of the host language. One notable exception are DSLs using Higher-Order Abstract Syntax [34], in which the binding in the host language is reflected in the embedded language.

8.3 Syntax

Our examples above use a specific syntax for describing transformations, solving rules, and type rules. We have made that syntax close to the one used by programming language experts when describing those elements in a compiler. However, this syntax might be too “low-level” for DSL authors. There is certainly room for improvement on how to expose the new compiler hooks.

9 Evaluation

This paper focuses on describing an architecture which enables DSL authors to provide domain-specific error messages. On its own, this does not guarantee that error messages get *better*. In a very extreme case, a DSL author could change every error message produced by their library to:

```
Your code is wrong, but I won't tell you why
```

This does not mean that domain-specific error messages are not highly desirable. GHC, for example, includes a very simple form of customization [42], which has been used to provide better error messages for lenses in the `silica` library, and records in the `vinyl` library, among others. Throughout this paper we have presented several examples which show actual improvements in the error messages of libraries comparable to those in the Haskell ecosystem (although in some cases the types have been simplified for the sake of conciseness.) We also showcase why we need each of the extensions to the compiler by means of different examples.

One remaining question is how one could *evaluate* the compiler architecture presented in this paper. This is a multi-faceted question, though. The Quique compiler, available at https://git.science.uu.nl/f100183/quique, provides an actual implementation of a Haskell-like language which follows the architecture presented in this paper. In particular, Quique features error contexts (but under a different name: “buckets”), and domain-specific solving and type rules. The type system in Quique is quite advanced. In particular, it includes higher-rank and impredicative polymorphism [35, 36], type families [37], and Generalized Algebraic Data Types (GADTs) [10]. This shows that our architecture is *implementable*.

Another side of the evaluation is whether our architecture is *better* than others which try to fulfill the goal of domain-specific error messages. Such a comparison is out of scope for this paper, as we focus on presenting new techniques to increase the expressiveness of error message customization. At the moment of writing there is no agreed set of benchmarks for "error reporting quality". Furthermore, in this paper we assume a compiler based on constraints, and the architecture would need to be reworked for type systems not based on this approach.

Finally, an *empirical* study could be performed to detect how useful customized error messages are for error reparation, and whether different groups of programmers (novice, experts) respond in measurably different ways to them. As in the previous case, such a study lies outside of the scope of the paper. Anyway, this is a highly subjective matter, so it is good to make the process as flexible and configurable as possible, which is exactly what we do.

10 Related work

The architecture discussed in this paper for domain-specific type error diagnosis builds upon previous work, mostly targeting functional languages as hosts. We do not give a comprehensive overview of the field, we just refer to those works closer to our approach.

Both Heeren *et al.* [13] and Wazny [27] introduce the idea of attaching error messages to constraints in order to allow domain-specific error reporting. As we have seen, DSL authors need a way to communicate those messages to the compiler. In this case, their solutions diverge. Wazny extends the notion of type signature, whereas Heeren *et al.* present specialized type rules. Specialized type rules subsume annotated type signatures, and for that reason we chose them as the basis for our domain-specific type rules as presented in Section 7.

We have discussed the importance of controlling solving and blaming order to produce good error messages. Hage and Heeren [38] acknowledge this challenge, and describe combinators for building trees of constraints. Due to the lack of recursive type rules, they cannot directly handle scenarios such as checking requirements of all elements in a list at the same time. They address the issue partially by means of *phase numbers* [13], which introduce a global ordering between all constraints of a binding group.

Some forms of domain-specific error reparation are discussed in [13], in particular sibling functions (function with different type but similar names which are easily confused) and repair directives. The transformations in Section 4 generalize the idea of directives to encompass other forms of reparation, such as type isomorphisms, which had not been integrated in a domain-specific setting until now.

We discussed how some domain-specific solving rules and axioms can be expressed within the host language itself. Type classes and type families in Haskell, for example, have a long history of usage to describe all kinds of DSLs. Some authors [39, 40] have advocated making these features more powerful in order to express more fine-grained invariants for a DSL. Chameleon [25] goes even further by giving DSL authors the ability to extend the Haskell language with user-defined rewriting rules.

The design space of what can be achieved with the integrated facilities of a programming language has also been explored. Kiselyov *et al.* [41] define an instance-less `Fail` type class in Haskell to indicate failure in solving. The `TypeError` constraint, as implemented in GHC [23], builds upon that idea by making the compiler aware of such a type class and changing slightly the error message in that case. In Scala it is possible to print a custom error message when the implicit search for values of a given trait is unsuccessful [24].

In the architecture described in this paper domain-specific information is threaded through all phases of the type checking and inference process. A completely different approach involves post-processing the type errors and the solving trace to inject domain-specific information. Christiansen [42] leverages the reflection facilities of the Idris language, allowing special functions to inspect the error messages from the compiler and change them. Plociniczak *et al.* [43] go a step further and enable exploration of the entire derivation tree.

Post-processing is clearly less invasive. If we want to apply our approach to an already existing compiler, the type checker needs to be re-architected. The downside of post-processing is the lack of control over the solving process. You can influence how an error is phrased or which expressions are blamed, but not what specific inconsistency was detected, as that would entail re-running parts of the solving process.

11 Conclusion

In this paper we have presented an architecture for a type checker which considers the problem of domain-specific error reporting as part of its core. In order to influence the wording of errors, the DSL author may introduce additional information during different phases of type checking. The result is context-dependent, that is, the place in which the error is detected may influence what is reported. In the future we want to extend this pipeline with conditional type rules [30].

We also want to explore how DSL authors use the facilities offered by our architecture or simpler ones such as

GHC's, and quantify the improvement in the ease of use of the DSL. Such a study should also reveal common patterns in the definition of custom type errors, providing the basis for a more user-friendly version of the techniques we present in this paper.

Acknowledgement: This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213). We would like to thank the Software Technology Reading Club at Utrecht for their very helpful comments, which have greatly helped to improve our presentation. The anonymous reviewers have helped a lot in shaping our discussion about limitations and evaluation of the approach.

References

[1] Voelter M., DSL Engineering – Designing, Implementing and Using Domain-Specific Languages, 2013

[2] Hudak P., Building domain-specific embedded languages, ACM Computing Surveys (CSUR) – Special issue: position statements on strategic directions in computing research, 1996, 28(4es), Article No. 196

[3] Marlow S., Haskell 2010 Language Report, 2010, https://www.haskell.org/onlinereport/haskell2010/

[4] Hage J., Domain specific type error diagnosis (DOMSTED), Technical Report UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014

[5] Damas L., Milner R., Principal type-schemes for functional programs, In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82), ACM, 1982, 207–212

[6] Lee O., Yi K., Proofs about a folklore let-polymorphic type inference algorithm, ACM Transactions on Programming Languages and Systems (TOPLAS), 1998, 20(4), 707–723

[7] Vytiniotis D., Peyton Jones S., Schrijvers T., Sulzmann M., OutsideIn(X): Modular type inference with local Assumptions, Journal of Functional Programming, 2011, 21(4-5), 333–412

[8] Swift Team, Type checker design and implementation, 2016

[9] Elm Team, Source code for Elm type checker, file Type/Solve.hs, 2018

[10] Sulzmann M., Wazny J., Stuckey P. J., A Framework for Extended Algebraic Data Types, In: Hagiya M., Wadler P. (Eds.), Functional and Logic Programming, FLOPS 2006, Lecture Notes in Computer Science, vol 3945, Springer, Berlin, Heidelberg 2006, 47–64

[11] Pottier F., Rémy D., The essence of ML type inference, In: Pierce B. C. (Ed.), Advanced Topics in Types and Programming Languages, MIT Press, 2005, 10, 389–489

[12] McAdam B. J., On the unification of substitutions in type inference, In: Hammond K., Davie T., Clack C. (Eds.), Implementation of Functional Languages, Lecture Notes in Computer Science, 1999, 1595, 137–152

[13] Heeren B., Hage J., Swierstra S. D., Scripting the type inference process, In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03), ACM, 2003, 3–13

[14] Heeren B. J., Top Quality Type Error Messages. Ph.D. thesis, Universiteit Utrecht, The Netherlands, September 2005

[15] Sulzmann M., Duck G. J., Peyton Jones S., Stuckey P. J., Understanding Functional Dependencies via Constraint Handling Rules, Journal of Functional Programming, 2007, 17(1), 83–129

[16] Heeren B., Leijen D., van IJzendoorn A., Helium, for learning Haskell, In: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell '03), ACM, 2003, 62–71

[17] Hage J., Heeren B., Heuristics for type error discovery and recovery, In: Horváth Z., Zsók V., Butterfield A. (Eds.), Implementation and Application of Functional Languages, IFL 2006, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, 4449, 199–216

[18] Stuckey P. J., Sulzmann M., Wazny J., Type processing by constraint reasoning, In: Kobayashi N. (Ed.), Programming Languages and Systems, Lecture Notes in Computer Science, 2006, 4279, 1–25

[19] Zhang D., Myers A. C., Toward general diagnosis of static errors, In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14), ACM, New York, NY, USA, 2014, 569–581

[20] Pavlinovic Z., King T., Wies T., Practical SMT-based type error localization, In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015), ACM, 2015, 412–423

[21] Loncaric C., Chandra S., Schlesinger C., Sridharan M., A practical framework for type inference error Explanation, In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016), 2016, 781–799

[22] McAdam B., How to repair type errors automatically, In: Trends in Functional Programming, Intellect Books, Exeter, UK, 2002, 87–98

[23] Diatchki I., Custom type errors, 2015, Available at https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors

[24] Scala Team, Docs for scala.annotation.implicitNotFound, 2015, Retrieved from https://www.scala-lang.org/api/2.12.7/scala/annotation/implicitNotFound.html

[25] Stuckey P. J., Sulzmann M., A theory of overloading, In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP '02), ACM, 2002, 167–178

[26] Gundry A., A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell, In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell 2015), ACM, 2015, 11–22

[27] Wazny J., Type inference and type error diagnosis for Hindley/Milner with extensions, Ph.D. thesis, University of Melbourne, Australia, 2006

[28] Serrano A., Hage J., Context-dependent type error diagnosis for functional languages, Technical Report UU-CS-2016-011, Department of Information and Computing Sciences, Utrecht University, 2016

[29] Serrano A., Hage J., Type error diagnosis for embedded DSLs by two-stage specialized type rules, In: Proceedings of the 25th European Symposium on Programming Languages and Systems (ESOP 2016), Springer-Verlag New York, 2016, 9632, 672–698

[30] Serrano A., Hage J., From attribute grammars to constraint handling rules, Technical Report UU-CS- 2016-010, Department of

Information and Computing Sciences, Utrecht University, 2016

[31] Lorenzen F., Erdweg S., Sound type-dependent syntactic language extension, In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016), ACM, 2016, 204–216

[32] Serrano A., Hage J., Lightweight Soundness for Towers of Language Extensions, In: Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017), ACM, 2017, 23–34

[33] De Koninck L., Schrijvers T., Demoen B., A flexible search framework for CHR, In: Schrijvers T., Frühwirth T. (Eds.), Constraint Handling Rules, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2008, 5388, 16–47

[34] Pfenning F., Elliott C., Higher-order abstract syntax, In: Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation (PLDI '88), ACM, 1988, 199–208

[35] Peyton Jones S., Vytiniotis D., Weirich S., Shields M., Practical type inference for arbitrary-rank types, Journal of Functional Programming, 2007, 17(1), 1–82

[36] Serrano A., Hage J., Vytiniotis D., Peyton Jones S., Guarded impredicative polymorphism, In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018), ACM, 2018, 783–796

[37] Chakravarty M. M. T., Keller G., Peyton Jones S., Associated type synonyms, In: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (ICFP '05), ACM, 2005, 241–253

[38] Hage J., Heeren B., Strategies for Solving Constraints in Type and Effect Systems, Electronic Notes in Theoretical Computer Science, 2009, 236, 163–183

[39] Heeren B., Hage J., Type class directives, In: Hermenegildo M. V., Cabeza D. (Eds.), Practical Aspects of Declarative Languages, PADL 2005, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2005, 3350, 253–267

[40] Morris J. G., Jones M. P., Instance chains: type class programming without overlapping instances, In: Proceeding of the 15th ACM SIGPLAN international conference on Functional programming (ICFP 2010), ACM, 2010, 45, 375–386

[41] Kiselyov O., Lämmel R., Schupke K., Strongly typed heterogeneous collections, In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Haskell '04), ACM, 2004, 96–107

[42] Christiansen D. R., Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages, Presented at TFP 2014, 2014, http://davidchristiansen.dk/drafts/error-reflection-submission.pdf

[43] Plociniczak H., Miller H., Odersky M., Improving Human-Compiler Interaction Through Customizable Type Feedback, 2014, https://infoscience.epfl.ch/record/197948/files/splash2014.pdf