Context-Dependent Type Error Diagnosis for Functional Languages

Alejandro Serrano

Jurriaan Hage

Technical Report UU-CS-2016-011
November 2016

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Context-Dependent Type Error Diagnosis for Functional Languages

Alejandro Serrano Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University
{A.SerranoMena, J.Hage}@uu.nl

Abstract

Customizable type error diagnosis has been proposed as a solution to achieve domain-specific type error diagnosis for embedded domain specific languages. A proven approach is to phrase type inferencing as a constraint-solving problem, so that we can manipulate the order in which constraints are solved, and associate domain-specific type error messages with specific constraints to be communicated to the programmer in case type checking fails. A major challenge in this area lies in scaling this idea up uniformly to a fully-featured (functional) language, that is, languages that go beyond the polymorphic λ-calculus.

In this paper, we show how within the general framework Constraint Handling Rules we can achieve such uniformity and generality, while at the same time providing the necessary type error customizability in a natural way. A proof-of-concept implementation is provided for a Haskell-like language, including support for type classes, GADTs and higher-ranked types. However, our approach applies to any language for which a constraint-based non-backtracking formulation of the type system is available.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Constraint and logic languages; D.3.4 [*Programming Languages*]: Processors – Compilers

Keywords Constraint Handling Rules, domain specific languages, custom error messages, type errors

1. Introduction

Type error customization, in whatever form, is an essential ingredient to support domain-specific type error diagnosis for *embedded domain-specific languages*. *Domain-specific languages* (DSL) are a widely used technique to solve problems in a particular domain in an effective way. DSLs can be either external, with its own tooling (Voelter 2013), or embedded, in which case the DSLs is embedded in a general-purpose host language (Hudak 1996). From the point of view of the host compiler, an embedded DSL is merely a library. Because of this, the host language compiler has no understanding of the domain, and will not be able to phrase type error messages in domain terms, leaking details of the encoding of the DSL into the host language in type error messages, putting users at a serious disadvantage (Hage 2014).

The `diagrams` library[1] is a DSL for describing drawings and animations embedded in Haskell. Its fundamental type is $QDiagram\ b\ v\ n\ m$ where b refers to the drawing backend (like SVG or PDF), v to the vector space, e.g., $V2$ and $V3$ (so it can deal with 2D and 3D pictures in a uniform way), n to the numeric type for coordinates (like $Double$ or $Integer$), and m to the type of annotations. Each of these parameters describes a very (domain-)specific aspect of a drawing.

Most drawing combinators impose restrictions on the combinations of parameters that can occur. One such combinator is

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

which puts one picture on top of the other, as long as they share the same backend, vector space, and so on. If only the vector spaces of the two arguments to $atop$ differ, we can add to the default message

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

a domain-specific diagnosis of the problem

```
‘atop‘ cannot combine diagrams from
  different vector spaces
```

Customization is also extremely helpful in education. During practice sessions, we can provide students with more context by referring to particular pages in the lecture notes, by restricting the applicability of very general libraries to specific use cases (reminiscent of the language levels in Racket (Tobin-Hochstadt et al. 2011) and the Helium compiler (Heeren et al. 2003)), or by contextualizing the message based on knowledge of the assignment they are working on.

In this paper we present a general methodology to support context-dependent custom error messages for a fully-featured (functional) language. The only requirement for the analysis is to be organised as a constraint-rewriting process (§ 3). This means it can be applied to, e.g., the type system implementation of the Glasgow Haskell Compiler (GHC) (Vytiniotis et al. 2011) and Apple's Swift compiler (Swift Team 2016).

In a constraint-based compiler, the solving order needs *not* be dictated, which reduces the bias inherent in standard substitution-tracking algorithms such as Algorithm $\mathcal{W}$ (Lee and Yi 1998; Hage and Heeren 2009). Although, the issue of bias has been addressed in the literature, many specifications and implementations of more advanced type systems still suffer from it. For example, keeping a

[1] Available at `http://hackage.haskell.org/package/diagrams`

global substitution for the entire solving process inherently bias the process towards the constraint leading to the substitution.

Buckets (§ 4) provide a way to assign custom error messages to failure scenarios by encoding the error diagnosis structure related to a set of constraints. Buckets also influence the order in which solving proceeds, which in turns helps in localizing error messages. By gaining control of the order in which solving proceeds, the bias problems are overcome. It is now the DSL author who chooses the priority on blaming, and not the compiler.

Throughout the paper we make use of Constraint Handling Rules (§ 2) – CHRs in short. We have decided to present our techniques using CHRs, because they provide us a well-understood language with clear syntax and semantics. The ideas, on the other hand, are not restricted to this formalism, and can be applied to any non-backtracking constraint rewriting formalism that supports priorities. The use of CHRs does ease the integration with other techniques to improve general error diagnosis (§ 5,§ 7).

In short, this paper offers the following contributions:

- We provide a technique to assign custom error messages and control constraint solving order, integrating earlier approaches of type inference directives (Heeren and Hage 2005) and GHC's special `TypeError` constraint (Diatchki 2015). We do so in the context of Constraint Handling Rules.
- We illustrate the viability of our contribution by applying our ideas to a small Haskell-like language. In addition, we have implemented a prototype for a larger language featuring GADTs and higher-rank polymorphism.

Non-goals. Our work provides a ground base which programmers can use to improve the quality of error reporting when developing domain-specific languages. It is not possible for us to directly evaluate how better error messages become, since they depend on the DSL author. If such a person decides to make type error messages less useful by always returning the text "You made a type error somewhere in your program but I won't tell you where", then there is nothing we/the underlying machinery can do about it.

Although in § 4 we present some syntax for integrating our technique in a Haskell-like language, this is only for illustration purposes. Clearly, the programmer must inform the compiler about the domain-specific information. But how to do so (syntax, same or different file) is not prescribed by our technique.

2. Constraint Handling Rules

Constraint Handling Rules (CHRs for short) form a high-level, declarative language originally designed to describe constraint solvers and simplifiers (Frühwirth 2009). The programmer specifies a set of *rules* which describe how to rewrite sets of constraints. The CHR engine then applies those rules exhaustively; the resulting set of constraints is thought of as the solution of the constraint problem.

The language of CHRs has three kinds of rules:

$$\begin{array}{lcl} H^r & \iff & g \mid B \quad \text{simplification} \\ H^k & \implies & g \mid B \quad \text{propagation} \\ H^k \setminus H^r & \iff & g \mid B \quad \text{simpagation} \end{array}$$

In each case, H^k, H^r and B are sets of constraints, called the heads and the body respectively. We use $\top$ to represent an empty set of constraints. In order for a rule to be applied, some constraints from the current set must match the heads, and the guard g must be satisfied. Rewriting depends on the kind of rule: with simplification rules the constraints H^r are replaced by B, in propagation rules the constraints B are added to the set but H^k are kept. Simpagation rules are a generalization of both: H^k constraints are kept and H^r are removed. In fact, we can see any CHR as a simpagation rule where the heads might be empty.

CHRs are applied non-deterministically. For a given initial constraint set, many different sequences of applications of rules are usually possible. The rule writer is responsible for guaranteeing confluence, that is, to ensure that the final result of a CHR query does not depend on the order in which rules are applied.

To illustrate how CHRs operate, we provide a simple example to find the largest number in a set S. There are two kinds of constraints: $\mathsf{in}(n)$ says that the number n belongs to S, and $\max(n)$ to represent that the largest observed number is n. The initial state is $\{\mathsf{in}(n) \mid n \in S\} \cup \{\mathsf{max}(-\infty)\}$. We need only one CHR:

$$\mathsf{in}(n) \setminus \mathsf{max}(m) \iff n > m \mid \mathsf{max}(n)$$

For $S = \{1, 3, 2\}$, the initial state is $\mathsf{in}(1), \mathsf{in}(3), \mathsf{in}(2), \mathsf{max}(-\infty)$. Non-deterministically, the solver may choose $\mathsf{in}(1)$ and $\mathsf{max}(-\infty)$ to apply the CHR to. Since the guard, $1 > -\infty$, is true, $\mathsf{max}(-\infty)$ is replaced by $\mathsf{max}(1)$. Since max has changed, it may try $\mathsf{in}(1)$ and $\mathsf{max}(1)$ again; now $1 > 1$ is false, so the rule does not apply. Note the importance of using $>$ instead of $\geqslant$ to ensure termination. The CHR solver may continue with $\mathsf{in}(3)$ and $\mathsf{max}(1)$, which leads to the replacement of $\mathsf{max}(1)$ with $\mathsf{max}(3)$. Finally, $\mathsf{max}(3)$ is matched with every other in constraint, but since no number is higher than 3 the solving process stops.

2.1 Rule priorities

Sometimes the soundness of a constraint solving procedure is only guaranteed if some rules are known to be applied before others. In other scenarios, control of the order in which CHRs are applied is necessary to increase performance. To achieve more control over the order rules are applied *rule priorities* have been added to the CHR language (De Koninck et al. 2007). In this case, rules have the general form:

$$H^k \setminus H^r \iff g \mid B \quad \mathsf{priority}\ e$$

The expression in e might involve information from the constraints in the heads, giving a great degree of control. The CHR engine then guarantees that applications of a rule with a higher priority are preferred. When more than one rule has the highest priority, the choice is again non-deterministic. Any ordered set can be used for priorities. In this paper we use natural numbers, where 0 has the highest priority and higher numbers represent lower priorities.

An example of the use of priorities is Dijkstra's shortest path algorithm.[2] We have three kinds of constraints: $\mathsf{edge}(A, B, n)$ describes a directed edge between nodes A and B with a weight n and $\mathsf{source}(A)$ states the starting point of the search. At the end, the constraint set is enlarged with $\mathsf{dist}(B, n)$ describing the shortest distance n found from the source to B.

$$\begin{array}{rcll} \mathsf{source}(A) & \iff & \mathsf{dist}(A, 0) & \mathsf{priority}\ 1 \\ \mathsf{dist}(B, d_1) \setminus \mathsf{dist}(B, d_2) & \iff & d_1 < d_2 \mid \top & \mathsf{priority}\ 1 \\ \mathsf{dist}(B, d), \mathsf{edge}(B, C, w) & \implies & \mathsf{dist}(C, d + w) & \mathsf{priority}\ d + 2 \end{array}$$

The priorities ensure that the edges are visited as Dijkstra's algorithm requires, without the need for any intermediate data structure. Note that the priority of the last rule depends on the value d encoded in its head. This ability shall prove itself helpful in the following sections.

3. Type inference using CHRs

Many analyses in a compiler are described via a *type system*, which assigns types to each construct (expression, declaration) in a program. The usual goal of a type system is restricting the kinds of

[2] This example is taken from (De Koninck et al. 2007).

Unif. variables		$\in$	$\alpha, \beta, \ldots$	
Skolem variables		$\in$	$a, b, \ldots$	
Type constructors		$\in$	$\mathsf{F}, \mathsf{G}, \ldots$	
Type variables	υ, ω	$::=$	$\alpha \mid a$	
(Mono-)Types	τ, ρ	$::=$	$\upsilon \mid \tau \to \rho \mid \mathsf{F}\,\tau_1 \ldots \tau_n$	
Constraints	Q	$::=$	$\tau = \rho$	
Type schemes	σ	$::=$	$\forall \overline{a}.\, \overline{Q} \Rightarrow \tau$	
Term variables		$\in$	$x, y, f, g, \ldots$	
Data constructor		$\in$	$F, G, \ldots$	
Expressions	e	$::=$	x	Variable
		$\mid$	$\lambda x.\, e$	Abstraction
		$\mid$	$e_1\, e_2$	Application
		$\mid$	$\mathsf{case}\ e\ \{\ \overline{F_i\, x_1 \ldots x_m \to e_i}\ \}$	Pattern matching
Environments	Γ	$::=$	$\epsilon \mid x : \sigma, \Gamma$	

Figure 1. λ-calculus with pattern matching

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

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

$$\frac{\alpha\ \text{fresh} \qquad \Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow C_1, C_2, \tau_1 = \tau_2 \to \alpha}\ \textsc{app}$$

$$\frac{\begin{array}{c}\beta, \overline{\gamma}\ \text{fresh} \qquad \Gamma \vdash e : \tau \rightsquigarrow C \\ \text{for each } F_i\, x_1 \ldots x_m \to e_i \text{ with } F_i : \forall \overline{a}.\, \overline{\tau_j} \to \mathsf{F}\,\overline{a}, \\ \overline{x_j : [\overline{a \mapsto \gamma}]\tau_j}, \Gamma \vdash e_i : \rho_j \rightsquigarrow C_j\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case}\ e\ \{\overline{F_i\, x_1 \ldots x_m \to e_i}\} \\ : \beta \rightsquigarrow C, \overline{C_j}, \overline{\rho_j = \beta}, \tau = \mathsf{F}\,\overline{\gamma}\end{array}}\ \textsc{case}$$

Figure 2. Constraint gathering for λ-calculus

behavior which are allowed at runtime. As the slogan says: "well-typed programs do not go wrong".

A traditional way to structure a type engine, the piece of software which checks adherence of a piece of code to the type system, is to traverse the Abstract Syntax Tree (AST) representing the code, performing some computation at each step. This path is taken by the classical $\mathcal{W}$ and $\mathcal{M}$ implementations of the Hindley-Milner type system (Damas and Milner 1982; Lee and Yi 1998). However, these syntax-directed algorithms are known to introduce some problems related to error reporting, in particular a bias coming from the fixed order in which they traverse the tree (McAdam 1999).

As motivated in the introduction, we choose instead to use constraints as the central component of our type engine. In a first phase the engine traverses the AST to *gather* the constraints that the assigned types must satisfy. Afterwards, a dedicated constraint *solver* is called, giving us back either a type assignment or a type error if the constraints are found to be inconsistent. This pipeline of gathering and solving is repeated per top-level binding in the source.

A constraint-based approach to typing is shown by Heeren *et al.* (Heeren et al. 2003; Heeren 2005) to be a *good* choice for better *error diagnosis*. The main reason for its aptness is that these systems do not impose a strict order on the whole process. Furthermore, once all constraints are gathered, the solver may have a more holistic view on the structure of the program. For example, it may decide to show a different error for a given identifier based on its use sites.

The main disadvantage of constraint-based approaches to typing is its performance. If a faster alternative is available, it can be used until an error is signalled. When an inconsistency is found, we go back to the constraint-based formulation. But we do not need to re-run the whole process: we just need to apply the constraint-based pipeline to the last top-level binding. This way we reach a good balance between performance and error diagnosis.

3.1 Constraint gathering

Constraint gathering is usually a syntax-directed process which traverses the code top-down with information about the environment and builds a set of constraints while going bottom-up. This process is represented as a judgment $\Gamma \vdash e : \tau \rightsquigarrow C$, where the environment Γ and the expression e are taken as inputs, whereas the type τ and the constraint set C are obtained as outputs.

As a running example, we shall use the λ-calculus with pattern matching given in Figure 1. This language is very similar to that used in the OUTSIDEIN(X) typing framework (Vytiniotis et al. 2011). Note that elements in the environment are typed with a scheme, which admits quantification and local constraints. Thus, we can have functions in the environment such as $id : \forall a.\, a \to a$. We just focus on the expression language here, an actual language would include signatures and declarations to introduce those type schemes.

The constraint gathering judgment for this language is given in Figure 2. In the case of a term variable, we need to freshen the variables in the type scheme before returning its type (remember that the judgment assigns a type to every expression, *not* a type *scheme*). In the case of abstraction and application, we just need to build or check the types based on its subexpressions. The most complex case is pattern matching, in which for every branch we need to build a new environment based on the type of the data constructor; at the end we need to check that the expression we match upon is compatible with all the patterns, and that the return types ρ_j of all branches are equal.

3.2 Implementing unification

Once constraints are generated, we give them to the solver, which either succeeds or fails if the constraint set is inconsistent. When the run is successful, it returns a *type assignment*, a mapping from type variables to types. Using that information we can assign a type to each expression in the program.

The constraint language we have introduced for the λ-calculus is enough to describe type assignments. We just need to ensure that in case of successful completion, all constraints are of the form $\alpha = \tau$, where α is a type variable and τ its assigned type, representing an *idempotent* substitution.

Most CHR implementations have a built-in unification operation. Although it is certainly the way to get the most performant implementation, it it at odds with our goal to achieve custom type error messages. The reason is that unification, as implemented in those engines, is a global operation, which might affect all constraints in the set. In contrast, we need certain type equalities to be visible only to a subset of the other constraints.

Our unification algorithm is given in Figure 3. It is similar to the algorithm given in (Martelli and Montanari 1982). Both algorithms check for syntactic equality and decompose terms (in this case, terms are those types headed by constructors). We signal an error whenever the type constructors do not match or the number of arguments differ. As usual there is an occurs check to forbid infinite types.

The main difference between the algorithm of Martelli and Montanari and ours lies in the handling of substitutions. Martelli and Montanari apply a global substitution when an equality of the form $\upsilon = \tau$ is found. Our algorithm does not thread substitutions to all other equalities at once, but one at a time. However, this has a caveat:

equalities with cyclic dependencies, such as $\alpha = \beta, \beta = \alpha$, loop indefinitely. The solution is to impose an arbitrary ordering $\prec$ on variables so that in every equality the smaller variable always comes first (Bachmair and Tiwari 2000). We call this step *orientation*. We extend the $\prec$ relation to all types by mandating type variables to be always smaller than types headed by a constructor.

Given a set of type equalities, if the solver returns a residual constraint set with no fail constraint, this resulting set can be turned into a type assignment. First of all, no equality of the form $\mathsf{F}\ \overline{\tau} = \mathsf{G}\ \overline{\rho}$ is present, because one of the three decomposition rules would match otherwise. Thus, we are left with just constraints of the form $\upsilon = \tau$. Furthermore, orientation and substitution ensures that the obtained type assignment is acyclic, or otherwise the occurs check would have produced a fail constraint.

3.3 Modeling type classes

Constraint Handling Rules have been used to explain how some programming language features work. In particular, the theory behind Haskell's type classes with functional dependencies has been modelled in this language (Sulzmann et al. 2007; Dijkstra et al. 2007). We recap here the main ingredients, and highlight the changes needed to integrate with our unification procedure. In Haskell, type class features appear in three different places: type class declarations, class instance declarations and type schemes.

During type checking, the question is whether a type τ is an *instance* of type class C. We also say that τ *belongs* to class C. In our CHR world, we represent it using a constraint $\mathsf{inst}(C, \tau)$.

First of all, we have *class declarations*, whose purpose is to introduce a type class C to the compiler. A type class may define *superclasses* $D_1, \ldots, D_n$ which every instance of C must implement. Haskell's syntax for these declarations is:

$$\textbf{class}\ (D_1\ \tau, ..., D_n\ \tau) \Rightarrow C\ \tau$$

In Haskell each type class come with an associated set of *methods*. In this paper we are only interested in the typing perspective of type classes, and thus omit further discussion of that topic.

Every declaration gives rise to a CHR rule. If we know that τ is an instance of C, then it must also be an instance of $D_1, \ldots, D_n$. Otherwise, the instance does not satisfy its superclass constraints. We can express this relationship via a propagation rule:

$$\mathsf{inst}(C, \tau) \implies \mathsf{inst}(D_1, \tau), \ldots, \mathsf{inst}(D_n, \tau)$$

The second kind of declarations are *instance declarations*. Instance declarations state that a given type τ is an instance of a type class C. The membership can be conditional on a context, that is, on instances of subparts of τ. Furthermore, these declarations introduce a notion of unicity: τ is an instance of C if and *only if* the context holds, there is no other way to make $\mathsf{inst}(C, \tau)$ hold.[3] Haskell's syntax for instance declarations is:

$$\textbf{instance}\ (D_1\ \rho_1, ..., D_m\ \rho_m) \Rightarrow C\ \tau$$

Each of these declarations is expressed as a CHR:

$$\mathsf{inst}(C, \tau) \iff \mathsf{inst}(D_1, \rho_1), \ldots, \mathsf{inst}(D_m, \rho_m)$$

As an example, Haskell's Prelude contains an Eq type class representing types with decidable equality. The instance for lists $[\tau]$ is only available if there is an instance for the element type τ.

$$\mathsf{inst}(Eq, [\tau]) \iff \mathsf{inst}(Eq, \tau)$$

Remember that in our CHR implementation, unification is not global, but rather we need to apply substitution constraint by constraint. In the algorithm in Figure 3 substitution is only threaded to other type equalities. We need a new rule to apply a substitution also over instance constraints:

$$\upsilon = \tau\ \setminus\ \mathsf{inst}(C, \rho) \iff \upsilon \in \mathsf{fv}(\rho)\ \mid\ \mathsf{inst}(C, [\upsilon \mapsto \tau]\rho)$$

In general, for every kind of constraint we have in our system, we need an explicit substitution rule. Implementors must be careful to ensure that substitution happens after orientation. Otherwise, cyclic dependencies between variables may cause the CHR solver to loop indefinitely.

Instance constraints also appear in type schemes. For example, the equality operator $(\equiv)$ uses the previously introduced Eq class:

$$(\equiv) :: \forall a.\ Eq\ a \Rightarrow a \to a \to Bool$$

We do not have to make any changes to accommodate for them, since our VAR rule in the gathering phase already allows any kind of constraints in schemes.

3.4 Recalling error conditions

We have seen how CHRs give us a unified language to discuss different compiler analyses. However, right now our failure mechanism is very primitive: just a special constraint fail. We would like to recall at least, which was the constraint that failed, so that the compiler may report something else than "your program is ill-typed".

In order to do so, we introduce a first *transformation* over CHRs. This will be a general scheme in this paper: a simple CHR system is progressively changed to account for new information.

In this case, we account for a constraint being part of an inconsistent set. Every time a set of constraints C leads to failure, we rewrite them to $\mathsf{fail}(C)$. If we are in the other scenario, because either the whole constraint set is satisfiable or the failure has not been found yet, the constraint is stated as $\mathsf{ok}(C)$.

Only if all constraints involved in a rule are still not involved in a failure, that is, they are all ok, we can apply a rule. Furthermore, the resulting constraints are still considered OK for solving. Thus, we need to transform all CHRs rules:

$$H_1^k, \ldots, H_n^k\ \setminus\ H_1^r, \ldots, H_\ell^r \iff g\ \mid\ B_1, \ldots, B_m$$

into corresponding rules:

$$\mathsf{ok}(H_1^k), \ldots, \mathsf{ok}(H_n^k)\ \setminus\ \mathsf{ok}(H_1^r), \ldots, \mathsf{ok}(H_\ell^r) \iff g\ \mid\ \mathsf{ok}(B_1), \ldots, \mathsf{ok}(B_m)$$

There is one exception: if in a rule we previously generated the fail constraint, we recall where the problem was. In that case, the transformation is from:

$$H_1^k, \ldots, H_n^k\ \setminus\ H_1^r, \ldots, H_\ell^r \iff g\ \mid\ \mathsf{fail}$$

into a CHR with annotated constraints:

$$\mathsf{ok}(H_1^k), \ldots, \mathsf{ok}(H_n^k)\ \setminus\ \mathsf{ok}(H_1^r), \ldots, \mathsf{ok}(H_\ell^r) \iff g\ \mid\ \mathsf{fail}(\{H_1^r, \ldots, H_\ell^r\})$$

The new rule has the same effect as the original with fail, because it disallows the H^r constraints to be used in further rule applications.

Note that this transformation does not change in any way which and when a set of constraints may interact using a specific rule. In addition, no rule applies to fail constraints, as usually happens with $\bot$ in CHRs. Thus, confluence and termination properties of the original set of CHRs are kept in their transformed version.

A simple improvement is to directly generate type error messages in the rules. We introduce a variation of the fail constraint, now with two arguments: one for the failing set of constraints, and one for the message. For example, we can report on unification failing because of different constructor heads with the rule:[4]

$$\mathsf{fail}(\{\mathsf{F}\ \overline{\tau} = \mathsf{G}\ \overline{\rho}\}) \iff \mathsf{fail}(\{\mathsf{F}\ \overline{\tau} = \mathsf{G}\ \overline{\rho}\}, \texttt{"Diff. constructors"}\ \mathsf{F}\ \texttt{"and"}\ \mathsf{G})$$

[3] If this restriction is relaxed, we speak of overlapping instances.

[4] We assume that string concatenation is built in the CHR language.

$$
\begin{array}{rcrcl}
\tau = \tau & \iff & & & \top \\
\mathsf{F}\ \tau_1\ \ldots\ \tau_n = \mathsf{F}\ \rho_1\ \ldots\ \rho_n & \iff & & & \tau_1 = \rho_1, \ldots, \tau_n = \rho_n \\
\mathsf{F}\ \tau_1\ \ldots\ \tau_n = \mathsf{F}\ \rho_1\ \ldots\ \rho_m & \iff & n \neq m & | & \mathsf{fail} \\
\mathsf{F}\ \tau_1\ \ldots\ \tau_n = \mathsf{G}\ \rho_1\ \ldots\ \rho_m & \iff & \mathsf{F} \not\equiv \mathsf{G} & | & \mathsf{fail} \\
\upsilon = \mathsf{F}\ \overline{\tau} & \iff & \upsilon \in \mathsf{fv}(\overline{\tau}) & | & \mathsf{fail} \\
\tau_1 = \tau_2 & \iff & \tau_2 \prec \tau_1 & | & \tau_2 = \tau_1 \\
\upsilon = \tau_1 \setminus \upsilon = \tau_2 & \iff & \upsilon \notin \mathsf{fv}(\tau_1, \tau_2) & | & \tau_1 = \tau_2 \\
\upsilon = \tau \setminus \omega = \rho & \iff & \upsilon \prec \omega, \upsilon \in \mathsf{fv}(\rho) & | & \omega = [\upsilon \mapsto \tau]\rho
\end{array}
$$

Figure 3. Unification as CHRs

With these changes, error reporting becomes part of the CHRs. This is important for our further developments, since now our transformations can depend on the presence of specific fail constraints to highlight the errors in the process. Note that this addition does not pose a threat to termination, since at most one step more per each of the fail constraints, which form a finite set.

4. Context-dependent custom error messages

The combination of error reporting constraints with rule priorities gives us a first recipe for custom error messages. In short, just override the generation of the two-argument version of fail for a specific constraint with a higher priority rule.

For example, we can give a custom message when the error involves a list. We need to handle the two symmetric cases.

$$
\begin{array}{l}
\mathsf{fail}(\{\tau = [\rho]\}) \iff \mathsf{fail}(\{\tau = [\rho]\}, \texttt{"A list was expected"}) \\
\mathsf{fail}(\{[\rho] = \tau\}) \iff \mathsf{fail}(\{[\rho] = \tau\}, \texttt{"A list was expected"})
\end{array}
$$

This approach is not fine-grained enough for neither education nor DSL purposes. Following the same example, the explanation of why we needed a list value is very different depending on the *context* in which such a constraint appears. When a student writes the following:

$$map\ (\lambda x \to x * x)\ 3$$

we could report as error:

```
The second argument to ‘map‘ must be a list
```

The example given in the introduction and featuring a drawing DSL is another example in which an error message can greatly benefit from customization based on the context in which it occurs. We only know that *V2* and *V3* represent vector spaces because they occur within a call to *atop*.

Given the usefulness of context-dependent error messages, we embark in a journey to support them in our system. A first approach is to move the messages from rules to constraints (Wazny 2006). The idea is that when a constraint leads to an inconsistency, the custom message is shown instead of the default one.

In order to account for this extra piece of information, we extend the ok constraint with a second argument, the error message. If the constraint leads to failure, we move this message as second argument to the corresponding fail constraint. For example, for the decomposition rule in unification:

$$\mathsf{ok}(\mathsf{F}\ \overline{\tau} = \mathsf{G}\ \overline{\rho}, msg) \iff \mathsf{F} \not\equiv \mathsf{G}\ |\ \mathsf{fail}(\{\mathsf{F}\ \overline{\tau} = \mathsf{G}\ \overline{\rho}\}, msg)$$

This new variation of ok constraints is generated during the constraint gathering phase.

The reader could surely think of dozens of different syntaxes to link constraints and error messages. As an example, we extend the syntax of our type schemes, optionally attaching a message to each of the constraints. The signature of map looks like:

$$
\begin{array}{l}
map\ ::\ \forall fn\ lst\ a\ b\ e. \\
\qquad fn = a \to b\ \mathsf{error}\ \texttt{"1st. arg. should be a fn."} \\
\qquad lst = [e]\ \mathsf{error}\ \texttt{"2nd. arg. should be a list"} \\
\qquad a = e\quad \mathsf{error}\ \texttt{"Fn and list do not coincide"} \\
\qquad \Rightarrow fn \to lst \to [b]
\end{array}
$$

Given that type signature, an application of map generates:

$$
\begin{array}{l}
\mathsf{ok}(fn = a \to b,\ \texttt{"1st. arg. should be a fn."}) \\
\mathsf{ok}(lst = [e],\quad \texttt{"2nd. arg. should be a list"}) \\
\mathsf{ok}(a = e,\quad \texttt{"Fn and list do not coincide"})
\end{array}
$$

4.1 Combining sets of constraints

In order for the presented scheme to apply, we need to describe a transformation that works for every possible CHR. The previous decomposition rule is just a very special case: only one constraint is consumed and the body is failure.

When the output is not a failure, it seems reasonable to propagate the error message to the newly produced constraints. Otherwise, the message is lost after one step of solving. One example of such a rule is given again by decomposition in unification, where we produce a new constraint per argument:

$$
\begin{array}{l}
\mathsf{ok}(\mathsf{F}\ \tau_1\ \ldots\ \tau_n = \mathsf{G}\ \rho_1\ \ldots\ \rho_n, msg) \\
\quad \iff \mathsf{ok}(\tau_1 = \rho_1, msg), \ldots, \mathsf{ok}(\tau_n = \rho_n, msg)
\end{array}
$$

This is the furthest we can take this idea, though. If the head contains more than one constraint, it is definitely not clear how to propagate the message. Take the substitution rule,

$$\mathsf{ok}(\upsilon = \tau, m_1)\ \setminus\ \mathsf{ok}(\omega = \rho, m_2) \iff \ldots$$

The body of the rule has to generate a messages from m_1 and m_2. We have three options: (1) we can use one of the messages, but in that case we are biasing the message to be shown, (2) we can concatenate both messages, but in large programs this might lead to huge messages which do not help the programmer, or (3) we can simply drop the message, in which case we lose information.

Even if we suppose that the messages are combined in the right way, DSL writers still do not have enough control over the solving process. This lack of control is due to the non-deterministic nature of CHRs, which might decide to apply matching rules in any order. If we take three constraints which form an inconsistent set,

$$\alpha = Int \quad \alpha = Bool \quad \alpha = Char$$

the type engine could take pair of those and signal an error. Thus, we have three possible messages:

```
Different constructors ‘Int‘  and ‘Bool‘
Different constructors ‘Int‘  and ‘Char‘
Different constructors ‘Bool‘ and ‘Char‘
```

This highlights that in order to have control over the error explanations, it is not enough to attach custom messages to constraints. We also need to think about what to do when several constraints are

combined, and how to influence the order in which constraints are considered by the type engine.

These problems are related to that of constraint *blaming*. Given a set of constraints known to be inconsistent, you want to point out a subset of those which can be held *responsible* for the inconsistency (Stuckey et al. 2006; Rahli et al. 2010; Zhang et al. 2015). In particular, you want to make this subset as informative as possible. Even though the techniques developed for blaming can be used to enhance error reporting, we believe that customization works better for the scenarios we want to support.

4.2 Introducing error buckets

We have seen that attaching custom messages to the constraints themselves does not work as expected for reporting. Our solution is to detach these two pieces of information: on one hand constraints are again seen as simple sets, and on other hand we have information about ordering and custom messages. More precisely, constraints are categorized into nested *buckets*, each of them holding an error message. This nesting structure gives us an ordering according to which constraints ought to be considered by the type engine.

Let us go back to the example of map from the beginning of the section. The bucket information for that function is depicted in Figure 4. The picture tells us that the type engine should first consider the constraints regarding fn being a function, and those regarding lst being a list, independently from each other, since they are in different buckets. If in any of those branches an inconsistency is found, the message leading its bucket is reported (it is possible that both are reported, too).

When any possible rule has been applied to the constraints over each argument, then the remainder constraints are allowed to interact between themselves and with the extra $a = e$ equality. This is done now under the umbrella of the outer bucket, so if an inconsistency is found at this stage, then the message to be reported is `"Fn and list do not coincide"`.

A nice property of error buckets is that once the solver gets to a certain bucket, it can safely assume that all nested constraints have completely been solved. Thus, some assumptions can be made about the state of solving at that time. In our example, we know that once the solver reaches the top-level bucket, either we have found an error, or we know that fn is indeed a function.

Definition. A *set of constraints with buckets* is a triple $\langle Q, \mathfrak{B}, \kappa \rangle$:

- Q is a set of constraints.
- $\mathfrak{B}$ is a set of buckets which form a rooted tree. That is, each bucket b has a single parent bucket, except for the root $\mathfrak{R}$.
- $\kappa : Q \to \mathfrak{B}$ assigns a bucket to each constraint in the set.

The conditions over $\mathfrak{B}$ allow us to define a partial order $b_1 \prec b_2$, which holds if b_2 is an ancestor of b_1, and an operation $b_1 \sqcup b_2$, which finds the nearest common ancestor of both buckets. It holds that $b \prec \mathfrak{R}$ for all buckets b.

Note that in this definition we do not have any information about error messages. Inside the compiler, we assign to each bucket label a certain error message, which is the one to be shown to the programmer. However, during the solving process, the only relevant information is the initial bucket and the ordering between them.

We lift the bucket assignment operation to sets of constraints:

$$\kappa\left(\{q_1, \ldots, q_n\}\right) = \bigsqcup \kappa(q_i)$$

The bucket reported by κ is the smallest one which contains all constraints. This gives us a precise answer to our question of how to combine the error messages (now, buckets) for several constraints: just take the $\sqcup$ of their corresponding buckets.

The ordering imposed by buckets needs to be taken into consideration during CHR solving.

Definition. Let Q be an inconsistent set of constraints with bucket $\kappa(Q)$. We say this set is *minimal* if there is no constraint $q \in Q$ that appears in another inconsistent set Q' such that $\kappa(Q') \prec \kappa(Q)$.

This minimality condition tells us we can only report an inconsistent set Q if none of the constraints in Q is involved in another error in a nested bucket. Note that for a given (set of) constraints, there can be more than one minimal set in which they are involved. In those cases, the usual error reporting mechanism built in the compiler is responsible for choosing one.

4.3 Implementing buckets using CHRs

Given a set of CHRs, we accommodate buckets by a transformation similar to § 3.4. Each constraint is now wrapped within ok or fail, depending on whether or not it has been found to be inconsistent. This wrapper includes a second argument which says in which bucket solving must take place.

Furthermore, we need to encode the relation between each bucket and its parent bucket. We do so by means of a new kind of constraint $\mathsf{bucket\text{–}parent}(c_1, c_2)$, which establishes that c_2 is the parent bucket of c_1. We assume that the root bucket is named $\mathfrak{R}$.

The constraint gathering process must be informed of the bucket in which each constraint must be generated. One possibility is to enlarge the language of type signatures. In the case of map, this variation looks like:

```
map fn lst
  :: bucket "Fn and list do not coincide" {
       bucket "1st. arg. should be a fn." {
         constraints fn, fn = a → b
       }, bucket "2nd. arg. should be a list" {
         constraints lst, lst = [e]
       }, a = e, result = [b]
     }
```

Using the same syntax, the bucket structure for the $atop$ function from the introduction is given in Figure 5.

A more flexible approach is to have specialized type rules which are triggered by the presence of certain kinds of expressions (Heeren et al. 2003). An advantage of this approach is that the choice of bucket may depend on the typing context of the expression (Serrano and Hage 2016). Regardless of the mechanism chosen to generate constraints, we assume that at the end of the gathering process all constraints are annotated with their bucket, and bucket parents are encoded as explained above.

Our implementation uses *bubbling* as a mechanism to guide solving. In short, we only allow constraints in the same bucket to interact. When we are sure that no more rewriting can be done in a given bucket, we "bubble up" the remaining constraints to its parent. This allows interaction with constraints in sibling buckets.

As in § 3.4, we only apply a rule if all constraints are still wrapped in ok. The novelty is that they must all live in the same bucket b, which is then propagated to the bodies. A general CHR of the form:

$$H_1^k, \ldots, H_n^k \setminus H_1^r, \ldots, H_\ell^r \iff g \mid B_1, \ldots, B_m$$

is transformed into:

$$\mathsf{ok}(H_1^k, b), \ldots, \mathsf{ok}(H_n^k, b) \setminus \mathsf{ok}(H_1^r, b), \ldots, \mathsf{ok}(H_\ell^r, b) \iff g \mid \mathsf{ok}(B_1, b), \ldots, \mathsf{ok}(B_m, b)$$

We stress the difference with the beginning of § 4.1: here the problem of flowing information from several heads does not appear, since we force everything to happen on the same bucket.

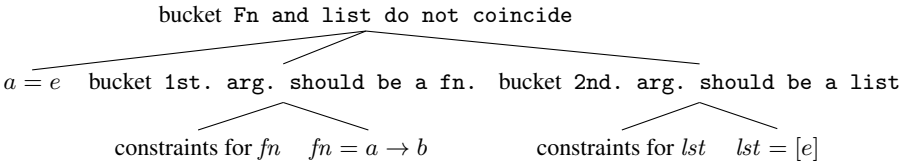

Figure 4. Error buckets for $map\ fn\ lst$

```
atop d1 d2
  :: bucket "Space prerrequisites are not met" {
       bucket "`atop` cannot combine diagrams with different annotation types" {
         bucket "`atop` cannot combine diagrams from different coordinate types" {
           bucket "`atop` cannot combine diagrams from different vector spaces" {
             bucket "`atop` cannot combine diagrams from different drawing backends" {
               bucket "First argument to `atop` is not a diagram" {
                 d1 = QDiagram b1 v1 n1 m1
               }, bucket "Second argument to `atop` is not a diagram" {
                 d2 = QDiagram b2 v2 n2 m2
               }, b1 = b2
             }, v1 = v2
           }, n1 = n2
         }, m1 = m2
       }, OrderedField n1, Metric v1, Semigroup m1
     }, result = QDiagram b1 v1 n1 m1
```

Figure 5. Type signature with bucket information for $atop\ d1\ d2$

When the rule ends in an error condition, a slight modification is needed. Instead of ok constraints, we recall the error in a fail constraint. The question on which bucket to report is now settled: if we detect the inconsistency when all constraints are currently living in bucket b, this is the bucket to be reported. The transformation takes a CHR of the form:

$$H_1^k, \ldots, H_n^k \ \setminus \ H_1^r, \ldots, H_\ell^r \iff g \mid \mathsf{fail}$$

and turns it into:

$$\mathsf{ok}(H_1^k, b), \ldots, \mathsf{ok}(H_n^k, b) \ \setminus \ \mathsf{ok}(H_1^r, b), \ldots, \mathsf{ok}(H_\ell^r, b) \iff g \mid \mathsf{fail}(\{H_1^r, \ldots, H_\ell^r\}, b)$$

With only these transformations, constraints in different buckets would never be able to interact with each other. The only way for constraints to interact is when they are in the same bucket, we introduce a CHR to bubble them up:

$$\mathsf{bucket\text{–}parent}(b_1, b_2) \ \setminus \ \mathsf{ok}(Q, b_1) \iff \mathsf{ok}(Q, b_2)$$

That is, if constraint Q lives in bucket b_1 and b_2 is its parent, then Q can be moved to b_2.

Unfortunately, the previous rule breaks some of our guarantees about buckets. Since CHR engines can non-deterministically choose which rule to apply next, it might decide to move all constraints up and up until they reach the root $\mathfrak{R}$ and only then begin with the "real" solving. But thanks to rule priorities, as introduced in § 2.1, we can schedule this rule to apply only when no other does. For example, by giving priority 2 to the bubbling rule and 1 to the rest.[5]

$$\mathsf{bucket\text{–}parent}(b_1, b_2) \setminus \mathsf{ok}(Q, b_1) \iff \mathsf{ok}(Q, b_2) \ \mathsf{priority}\ 2$$

This is, however, still not enough, because we need to schedule bubbling to happen in the right order. Consider this situation:

- Constraint Q_1 lives in bucket b_1, and Q_2 in b_2.
- b_1 and b_2 are different, but have the same parent b.
- Bucket b is not the root, so it has a parent, say b'.

With the bubbling rule a possible trace of execution is that Q_1 is bubbled up to b and then to b'. Because Q_2 is in b_2, no interaction can take place. Then Q_2 is bubbled twice too, to b'. At the end, Q_1 and Q_2 interact in bucket b', but we wanted them to do so in b.

The solution is simple: bubbling of more nested buckets happen before less nested ones. In this case, we must give priority to Q_2 moving to b in contrast to Q_1 moving from b to b'. We achieve so by changing the priority of the rule depending on the bucket.

Definition. The *height* of a constraint b, denoted $\mathfrak{h}(b)$, is the length of the shortest path from the root $\mathfrak{R}$ to b. More formally, we say b' is an *ancestor* of b if either (1) b' is the parent of b, or (2) b' is an ancestor of the parent of b. Then:

$$\mathfrak{h}(b) = \#\{b' \mid b' \text{ is an ancestor of } b\}$$

where $\#S$ denotes the cardinality of a set S.

The *height* of a set of buckets $\mathfrak{B}$, $\mathfrak{h}_\mathfrak{B}$ is defined as the maximum height of any of its buckets. This corresponds to the height of the tree representing the buckets.

The final version of the bubbling rule uses the nesting level to prefer bubbling constraints lower in the tree over ones closer to the root bucket. The constant 2 ensures that in any case bubbling has less priority than rules from the type system.

$$\mathsf{bucket\text{–}parent}(b_1, b_2) \setminus \mathsf{ok}(Q, b_1) \iff \mathsf{ok}(Q, b_2) \ \mathsf{priority}\ 2 + \mathfrak{h}_\mathfrak{B} - \mathfrak{h}(b_2)$$

This ensures that no constraint is moved from b_1 to its parent b_2 until all constraints that live in buckets nested inside b_1 have been

[5] Remember smaller numbers mean higher priority.

bubbled up. The combination of this fact with the only interaction of constraints in the same bucket achieves the minimality condition.

Theorem 1. *Given a set of* bucket–parent *constraints describing a tree structure on buckets.*

1. *If the original set of CHRs is confluent, any two traces of the modified CHRs starting with the same initial set also comes to a equivalent state, modulo the buckets in the* fail *constraints.*
2. *If the original set of CHRs terminate for a given initial set of constraints, the modified CHRs also terminate for it.*

Proof. The key point is noticing that each trace of execution using the modified CHRs corresponds to a trace of execution using the original set, plus some applications of the bubbling rule.

We can only apply the bubbling rule a finite number of times, corresponding to the number of nodes in the bucket–parent tree. Thus, if the trace is finite with the original CHRs, it is so with the modified. Hence, (2) holds.

On the same vein, given two traces of the modified CHRs starting with the same initial set of constraints, the corresponding traces using the original set must be equivalent, because of the confluence of the original CHRs. In particular, if at the final state no fail constraint is in the current set, the final states with the modified CHRs must also be equivalent. In the case in which fail appears, we cannot rule out the possibility of finding the inconsistency on different moments. For that reason, the buckets appearing as arguments in fail constraints might be different. □

The other important property we want for our CHR implementation of buckets is *minimality*. That is, the inconsistent sets it finds are always minimal. Alas, it is not possible to guarantee this fact in general. The reason is that there might exist an inconsistent set of constraints whose inconsistency is not detected (ever or at the right time) by the rules. This refrains the modified CHRs from finding the correct bucket to report.

Definition. We say that a set of CHRs is *inconsistency-exhaustive* if given an inconsistent set of constraints as input, they are guranteed to generate a fail constraint.

Theorem 2. *Assume a given set of constraints with buckets and a set of CHRs to which the transformation is applied. If the CHRs are inconsistency-exhaustive, then every argument to* fail *is a minimal inconsistent set.*

Proof. Suppose that there exists an argument to fail, namely I, which is not a minimal inconsistent set. Then there exists another inconsistent set J such that $\kappa(J) \prec \kappa(I)$, but which has not been found as such by the CHRs. This contradicts the definition of inconsistency-exhaustive. □

4.4 Obtaining custom messages from buckets

A final step is to make our failure message appear directly as the residual of rules. One additional fact is needed: the mapping between buckets and messages, which we can keep as a new kind of constraint, $\mathsf{bucket\text{–}message}(c, msg)$. The CHR

$$\mathsf{bucket\text{–}message}(c, msg) \setminus \mathsf{fail}(Q, c) \iff \mathsf{fail}(Q, msg)$$

replaces the bucket from the failure with the message associated to that bucket. Once the message is set, we no longer need to recall the bucket in which the inconsistency was found, and discard it. Whenever an inconsistency shows up in the root bucket, $\mathfrak{R}$, and we have no custom error message to give, then we use the default compiler messages.

5. Custom messages for CHR rules

As mentioned in the introduction, our work is not the first in the area of custom type error messages. Our framework can cope with many of the previous approaches by reformulating them as CHRs.

GHC, since version 8, includes a facility for custom type errors via the `TypeError` constraint (Diatchki 2015). Every time this constraint is found during solving, the compiler shows the custom message encoded inside of it, and enters an error state. One use case is the flag erroneous instances of a type class:

$$\textbf{instance}\ \mathit{TypeError}\ \texttt{"Cannot show functions"} \Rightarrow \mathit{Show}\ (a \to b)\ \textbf{where}\ \ldots$$

Operationally, when the type engine is faced with a $\mathit{Show}\ (a \to b)$ instance, it rewrites it to a $\mathit{TypeError}$ constraint. When it finds a $\mathit{TypeError}$, the message is shown.

From the CHR perspective, the important part is to signal a $\mathit{TypeError}$ constraint as a fail. There is one choice to be made: does the custom error message encoded in the $\mathit{TypeError}$ take precedence over the custom message coming from the bucket. If the $\mathit{TypeError}$ supersedes the current bucket, the rule reads:

$$\mathsf{ok}(\mathsf{inst}(\mathit{TypeError}, msg), c) \iff \mathsf{fail}(\{\mathsf{inst}(\mathit{TypeError}, msg)\}, msg)$$

On the other hand, we might decide to take the bucket into account:

$$\mathsf{ok}(\mathsf{inst}(\mathit{TypeError}, msg), c) \iff \mathsf{fail}(\{\mathsf{inst}(\mathit{TypeError}, msg)\}, c)$$

(note the difference in the second argument to fail). In this case, the $\mathit{TypeError}$ message is only shown if the constraint reaches the root:

$$\mathsf{fail}(\{\mathsf{inst}(\mathit{TypeError}, msg)\}, \mathfrak{R}) \iff \mathsf{fail}(\{\mathsf{inst}(\mathit{TypeError}, msg)\}, msg)$$

Along the same lines, more complex policies for dealing with the precedence of messages can be developed. The important point to get from this discussion is that our CHR machinery can encode the operation of `TypeError` as found in GHC.

Several kinds of directives involving custom information about type classes are described in (Heeren and Hage 2005). For example, the **never** directive instructs the compiler that no instance of a specific form may exist for a type class.

$$\textbf{never}\ \mathit{Show}\ (a \to b)\ \texttt{"Cannot show functions"}$$

And the **disjoint** directive specifies that no type can be an instance of two classes at the same time. In the Haskell world, $\mathit{Integral}$ and $\mathit{Fractional}$ are examples of such a pair of classes.

$$\textbf{disjoint}\ \mathit{Integral}\ \mathit{Fractional}\ \texttt{"No type can be both integer and fraction"}$$

The **never** directive is in fact very similar to the $\mathit{TypeError}$ one, and the solutions are similar to: as earlier, $\mathsf{inst}(\mathit{Show}, a \to b)$ should simply rewrite to an error. But instead of having an intermediate step with $\mathit{TypeError}$, we go directly to fail:

$$\mathsf{ok}(\mathsf{inst}(\mathit{Show}, a \to b), c) \iff \mathsf{fail}(\{\mathsf{inst}(\mathit{Show}, a \to b)\}, \texttt{"Cannot show functions"})$$

The same discussion about precedence of the message of the directive with respect to buckets apply here. The rule shown is the one corresponding to precedence of the **never** directive. Disjointness can be encoded similarly as a CHR:

$$\mathsf{ok}(\mathsf{inst}(\mathit{Integral}, \tau), c), \mathsf{ok}(\mathsf{inst}(\mathit{Fractional}, \tau), c) \iff \mathsf{fail}(\{\ldots\}, \texttt{"No type can be both ..."})$$

The various examples of this section show a similar pattern. First, we have a rule whose consequent is a fail constraint. In other words, we can identify a constraint (or more than one) which

are inconsistent. We want that rule to incorporate a custom error message, usuallu coming from the definition of the rule from programmer input. The solution is to have a custom Constraint Handling Rule. This pattern was already described in Section 6.2 of (Wazny 2006). In that work, the underlying technique for implementing type systems coincides with ours, CHRs.

None of the two approaches to obtain custom messages, namely buckets and rules, are a generalization of the other. They are rather complementary, in fact. Custom CHRs allow to express domain-specific information which holds for every use of a certain type or constraint, irrespective of its context or position in the AST. Buckets information, on the other hand, is always generated for specific expressions. The combination gives a powerful set of tools to customize error messages, but some care needs to be given to decide precedence of custom error messages from buckets with respect of those from rules.

6. Implementation

We have used our techniques to support context-dependent error messages in a prototype Haskell-like language. The implementation is available at `https://git.science.uu.nl/f100183/quique`.

In addition to the type system features described in this paper, our prototype also supports Generalized Algebraic Data Types (Xi et al. 2003) and higher-rank polymorphism. Both features have in common the use of *local reasoning*, that is, solving constraints under different assumptions. These different environments correspond, for example, to different branches in a GADT pattern match.

Most compilers making use of local reasoning (Vytiniotis et al. 2011; Sulzmann et al. 2006; Pottier and Rémy 2005) divide its operation between a solving stage and a driver which takes care of adjusting the environment in each case. This approach introduces bias, e.g., when pattern matching on a GADT, the blame for a type inconsistency is placed on the arms of the case statement, not on the matched expression. Furthermore, it goes against our philosophy of a single CHR-based resolution stage.

Our solution involves introducing annotations for each constraint recording the environment in which they originate. The annotation on a constraint mandates which other constraints they may interact with, and whether variables need to be considered as unifiable or as Skolem rigid constants. In that way, the restrictions imposed by local reasoning are visible in the CHRs. As a result, an external driver is no longer needed. In turn, this means that we can make use of our technique for context-dependent type error messages even when local reasoning is used. Unfortunately, due to a lack of space, we cannot present here the procedure to annotate constraints and the modifications needed for CHRs.

7. Related work

Customizable error messages. Apart from the work discussed in § 5, a few other compilers provide the ability to change the default error messages generated by the compiler.

In our system, customization is integrated with the analysis itself: buckets guide the solving process and type errors detected during solving influence the constraints in a bucket at a given time. Another possibility is to post-process the output of the typing process, (Plociniczak et al. 2014; Plociniczak 2016) takes the typing derivation constructed by the Scala compiler which can then be programmatically explored. In that way, common patterns can be detected to produce domain-specific messages.

Post-processing has been applied in the context of the dependently typed programming languages Idris (Christiansen 2014): in that case the programmer can inspect the erroneous code and the type error generated for it by means of reflection. If desired, the default error message can be replaced by a custom message.

An approach based on post-processing has the advantage of not requiring any change to the resolution process in the analysis, only some form of instrumentation to generate the output. However, it does require deep knowledge of the way the compiler represents its analysis trace, which might change whenever a new version is released. Moreover, it is not possible to influence the solving process itself, something we can model by our use of buckets.

Other approaches to enhance error diagnosis. The problem of giving informative and helpful messages when the compiler detects an error has been approached from many other directions. We discuss a few of the more recent advances below.

Several approaches work on a graph representation of the type constraints. Heuristics geared to detecting particular kinds of mistakes, are implemented by traversing such a graph, in order to find the most likely error from that inconsistent set of constraints. In (Zhang and Myers 2014) Bayesian techniques are applied to OCaml and a flow analysis, which is later extended in (Zhang et al. 2015) to cope with the local reasoning needed in GHC. A representation of type equalities called type graphs is used in (Hage and Heeren 2006). In that case, heuristics are used to select which constraint is to be blamed, and through that which type error message is returned.

Many approaches try to focus on giving one single location as responsible for an error. *Slicing* however tries to inform the user of every place in a piece of code which contribute to an inconsistency. The aim is to give the programmer a complete view of the problem, and remove any bias the solver may have to blame specific kinds of problems or locations. Skalpel (Rahli et al. 2015; Haack and Wells 2004) is a type error slicer for Standard ML based on constraints. Minimal unsatisfiable subsets are used in (Stuckey et al. 2006) to provide smaller slices of the program which contribute to an error in a Haskell program. The set of possible sources of an error can also be narrowed using programmer interaction. This is the idea behind type debuggers (Chitil 2001; Tsushima and Asai 2013).

Counter-factual typing (Chen and Erwig 2014) tries to keep solving while keeping track of erroneous expressions found during the process. In order to do so, it keeps different variations of the typing derivations, and then extracts error messages from those variations which lead to inconsistencies.

8. Conclusion and future work

This paper presents a techniques to integrate context-dependent custom type error messages into a type system implementation defined using CHRs. The transformations are mechanical and impose no conditions over the initial CHRs, so it is widely applicable.

One disadvantage of our method is that unification has to be entirely based on CHRs. This is sound, but not rather inefficient. We have already pointed out that this problem can be lessened by only using the most expensive CHR-based solving when an error is detected. Nevertheless, we want to explore other options, in particular keeping track of a global substitution from which each constraint only has a partial view. In our case, this partial view would depend on the bucket holding the constraint.

For some type systems, such as Swift's (Swift Team 2016), backtracking is needed to explore different possible solutions. A variant of CHRs, CHR$^\vee$ (Abdennadher and Schütz 1998; Koninck et al. 2008) includes disjunction to deal with such scenarios. We aim to explore which are the changes needed to integrate our techniques into this larger formalism.

Acknowledgments

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 also like to thank everybody in the Computer Science Reading Club at Utrecht University for their helpful comments in early drafts of this paper.

References

S. Abdennadher and H. Schütz. CHR$^\vee$: A Flexible Query Language. FQAS '98, pages 1–14. Springer-Verlag, 1998.

L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, pages 64–78. Springer-Verlag, 2000.

S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594. ACM, 2014.

O. Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. ICFP '01, pages 193–204. ACM, 2001.

D. R. Christiansen. Reflect on Your Mistakes! Lightweight Domain-Specific Error Messages. *Presented at TFP 2014*, 2014.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982.

L. De Koninck, T. Schrijvers, and B. Demoen. User-definable Rule Priorities for CHR. PPDP '07, pages 25–36. ACM, 2007.

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

A. Dijkstra, G. van den Geest, B. Heeren, and S. D. Swierstra. Modelling Scoped Instances with Constraint Handling Rules. Technical report, Department of Information and Computing Sciences, Utrecht University, 2007.

T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 1st edition, 2009.

C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

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

J. Hage and B. Heeren. Heuristics for Type Error Discovery and Recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2006.

J. Hage and B. Heeren. Strategies for Solving Constraints in Type and Effect Systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183, Apr. 2009.

B. Heeren and J. Hage. Type Class Directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, PADL'05, pages 253–267, Berlin, Heidelberg, 2005. Springer-Verlag.

B. Heeren, J. Hage, and S. D. Swierstra. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 3–13. ACM, 2003.

B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.

P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996.

L. Koninck, T. Schrijvers, and B. Demoen. Constraint handling rules. chapter A Flexible Search Framework for CHR, pages 16–47. Springer-Verlag, 2008.

O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, July 1998.

A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, Apr. 1982.

B. J. McAdam. On the Unification of Substitutions in Type Inference. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 137–152. Springer Berlin Heidelberg, 1999.

H. Plociniczak. *Decrypting Local Type Inference*. PhD thesis, IC, Lausanne, 2016.

H. Plociniczak, H. Miller, and M. Odersky. Improving Human-Compiler Interaction Through Customizable Type Feedback. 2014.

F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. URL `http://cristal.inria.fr/attapl/`.

V. Rahli, J. B. Wells, and F. Kamareddine. A constraint system for a SML type error slicer. Technical report, Heriot-Watt University, 2010.

V. Rahli, J. B. Wells, J. Pirie, and F. Kamareddine. Skalpel: A type error slicer for standard ML. *Electr. Notes Theor. Comput. Sci.*, 312:197–213, 2015.

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

P. J. Stuckey, M. Sulzmann, and J. Wazny. Type Processing by Constraint Reasoning. In N. Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2006.

M. Sulzmann, J. Wazny, and P. J. Stuckey. A Framework for Extended Algebraic Data Types. FLOPS'06, pages 47–64. Springer-Verlag, 2006.

M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.*, 17(1):83–129, Jan. 2007. ISSN 0956-7968.

Swift Team. Type checker design and implementation, 2016. URL `https://github.com/apple/swift/blob/master/docs/TypeChecker.rst`.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141. ACM, 2011.

K. Tsushima and K. Asai. *An Embedded Type Debugger*, pages 190–206. Springer Berlin Heidelberg, 2013.

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

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

J. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, University of Melbourne, Australia, 2006.

H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. POPL '03, pages 224–235. ACM, 2003.

D. Zhang and A. C. Myers. Toward General Diagnosis of Static Errors. POPL '14, pages 569–581, New York, NY, USA, 2014. ACM.

D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones. Diagnosing type errors with class. PLDI 2015, pages 12–21. ACM, 2015.