Extended Abstract: Improving Error Messages for Dependent Types

Joseph Eremondi
University of British Columbia
jeremond@cs.ubc.ca

Wouter Swierstra
Utrecht University
w.s.swierstra@uu.nl

Jurriaan Hage
Utrecht University
j.hage@uu.nl

1 Introduction

Dependently typed languages allow programmers to establish the correctness of their code, accessing the full power of higher-order logic via the Curry-Howard correspondence. However, a major barrier to their widespread adoption is their complexity. Since they impose a rigid type discipline, a significant portion of development time is spent reading, understanding, and responding to compiler error messages.

For Hindley-Milner style functional languages, such as Haskell or ML, several techniques have been developed to improve the quality of error messages. Our work adapts these techniques to dependently typed languages. We present **replay graphs**, which provide a representation of a unification algorithm run as a graph, allowing for the use of heuristics to generate error messages and repair hints, and **counterfactual unification**, which makes unification resistant to *bias*, so that when conflicting assumptions are encountered, the first one is not necessarily assumed to be correct.

2 Error Message Goals and Concepts

Before describing how to improve dependently typed error messages, we first need to look at what improvements we are seeking to achieve.

2.1 Error Location and Cause

A major aspect of message generation is choosing one or more source-code locations to associate with the error. We wish to report all locations that the programmer must inspect to make a repair. Additionally, we would like error messages to indicate the *cause* of the error: the specific mistake whose removal will cause the program to typecheck. Even better is to suggest a *repair*: the change that must be made to correct the error.

For the Agda code in Listing 1, the function `myOp` performs arithmetic on a pair of numbers and another number. When we try to fold `myOp` over a list containing number-boolean pairs, we get a type error. The reported message locates the error at `myList`. However, it is missing crucial information, namely that we are expecting a number because of the type of `myOp`. Thus, both locations are relevant to the error. This cause is hidden by the fact that this constraint is induced by the implicit arguments to `foldr`.

TYDE'18, September 27, 2018, St. Louis, MO, United States
2018.


```
foldr : {A : Set} {B : Set} → (A → B → B) → B → List A → B
myOp : ℕ × ℕ → ℕ → ℕ
myList : List (ℕ × Bool)
myVal : ℕ
myVal = foldr myOp 0 myList
--Bool !=< ℕ of type Set
--when checking that the expression myList has type List (ℕ × ℕ)
```

Listing 1. Error with multiple relevant locations

```
myZipWith : {A B : Set} → ((A × A) → B) → List A → List A → List B
myVal1 = myZipWith proj1 (1 ∷ 2 ∷ []) (true ∷ false ∷ [] )
-- Bool !=< ℕ of type Set
-- when checking that the expression true has type ℕ
myVal2 = myZipWith proj1 (true ∷ false ∷ [] ) (1 ∷ 2 ∷ [])
-- ℕ !=< Bool of type Set
-- when checking that the expression 1 has type Bool
```

Listing 2. Bias in error messages

2.2 Left to Right Bias

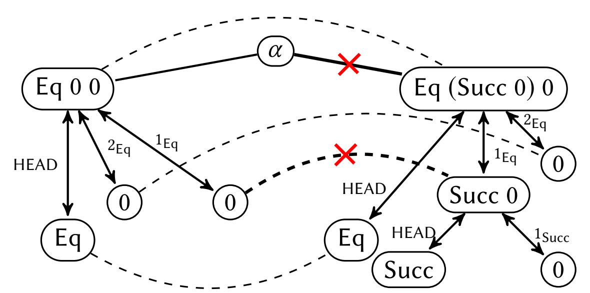
Another cause of unsatisfactory messages is *bias*. When program points imply different types for an expression, whichever the typechecker sees first is often assumed to be correct, regardless of which is more likely to be the correct type.

Consider the Agda code in Listing 2. In the first example, `true` is reported as ill-typed, and the correct type is assumed to be a number, even though switching the types of either list will remove the error. When the order of the lists is changed, instead `1` is the error location, and the correct type is assumed to be `Bool`, showing bias.

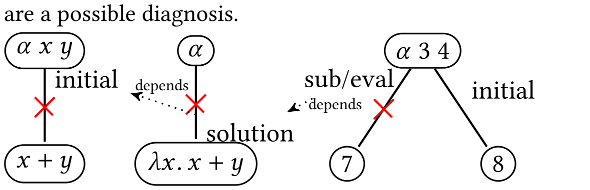
3 Higher Order Unification

Type inference plays a key role in the usability of dependent types. For example, most dependently typed languages have no explicit parametric polymorphism. The familiar type $\forall X. T$ is instead simulated with the dependent type $(X : \mathrm{Set}) \rightarrow T$. To enable hygenic use of polymorphism, functions are explicitly instantiated with *program metavariables*, holes whose value is determined during compilation. Typechecking becomes a constraint satisfaction problem.

To solve these constraints, we need a *higher-order unification* algorithm. A higher-order unification problem consists of a set of metavariables $\alpha_1 \ldots \alpha_n$ and a set of problems of the form $\forall \Gamma.\, S \equiv T$, where S and T are terms, and Γ stores a list of typed free program variables. In a dependently typed language, types and values depend on each other: S and T need not be types, but may contain functions, applications, and any other terms from our language. A solution consists



(a) Replay graph for $\{\alpha \equiv \mathrm{Eq}\ 0\ 0, \alpha \equiv \mathrm{Eq}\ (\mathrm{Succ}\ 0)\ 0\}$. Dashed edges are implied by equality paths, and the crossed-out edges are a possible diagnosis.



(b) Replay graph for $\{\forall x, y.\ \alpha\ x\ y \equiv x + y, \alpha\ 3\ 4 \equiv 8\}$. Inversion gives $\alpha := \lambda x.\ x + y$, so $\alpha\ 3\ 4$ gets an evaluation edge to 7. The crossed-out edges are a possible diagnosis.

Figure 1. Example replay graphs

of a value for each α_i such that the sides of each equation are equal up to $\beta\eta$ reduction. While higher order unification is undecidable in general, algorithms exist [1, 4, 5] deciding large enough fragments that it can be used in practice.

4 Dependent Type Error Strategies

4.1 Replay Graphs

Helium [6–8] is a Haskell compiler that facilitates high quality error message generation through *constraint graphs*. Each subterm is represented by a node in the graph, with undirected edges denoting equality. Directed edges connect terms and their subterms, and implicit edges are added between the subterms of connected terms (e.g. $\{S \rightarrow S', T \rightarrow T'\}$ induces edges $\{S, S'\}$ and $\{T, T'\}$).

This allows us to diagnose an error by choosing edges whose removal disconnects all non-equal nodes in the graph. Each edge corresponds to a source location, and the graph is not biased by the order in which constraints are added. Heuristics can be used to generate error messages, considering all relevant program points, and the graph can be easily edited, so that heuristics can search for potential repairs.

This approach addresses the issues from Section 2, but it fails in a dependently-typed setting. Typechecking involves evaluation of terms, and higher order problems often must be transformed before they can be solved. The graph approach handles injective constructors, but if, for example, we have, $(\lambda x.\ \lambda y.\ 0)\ 0\ 0 \equiv (\lambda x.\ \lambda y.\ x)\ 0\ 1$, then we cannot conclude that $(\lambda x.\ \lambda y.\ 0) \equiv (\lambda x.\ \lambda y.\ x)$, or that $1 \equiv 0$.

```
plus :: Nat -> Nat -> Nat
pNPlus0isN :: forall n :: Nat . Eq Nat (plus n 0) n
let succPlus = (\n -> pNPlus0isN (Succ n))
      :: forall n::Nat. Eq Nat (Succ n) (plus (Succ n) 0)
--    HINT: Rearrange arguments to match
--    Eq Nat (Succ x) (plus (Succ x) 0)
--    to Eq Nat (plus (Succ x) 0) (Succ x)
```

Listing 3. Message with repair hint

To account for these difficulties, we compromise. Constraints are solved using a higher-order unification algorithm, which decomposes problems, performs substitution and evaluation, and emits metavariable solutions. For each intermediate problem $S \equiv T$ or solution $\alpha := T$, we add the corresponding edge to our graph.

To account for evaluation, terms containing function abstractions or applications are stored in our graph. If a term T is in our graph, and α is a metavariable in T, then when a solution $\alpha := S$ is emitted, we add a new node T' for the evaluated form of $[X \Mapsto S]T$, and add an edge $\{T, T'\}$. Thus, we obtain a path between T and its successive evaluations as more solutions are discovered.

When typechecking is complete, we are left with a *replay graph* tracing the steps unification took. This can be analyzed using the same techniques as Helium, allowing for error messages and repair hints to be generated. Example replay graphs are shown in Figure 1.

4.2 Counter-Factual Unification

Some bias is still present in replay graphs. Because dependent typechecking performs evaluation at compile-time, when a solution $\alpha := t$ is generated, t is substituted for all occurrences of α. Since the first possible solution for α is the only one substituted, we are biased by the order in which we process constraints.

To rectify this, we employ *counter-factual unification*, based on the concepts of counter-factual typing [2] and the *choice calculus* [3] to concisely represent sets of terms.

The core idea is, whenever $\alpha := t$ is generated as a solution, we instead generate $\alpha := C\langle t, \alpha' \rangle$, where α' is fresh. Here, $C\langle t, \alpha' \rangle$ is the variational expression with choices t and α'. After solving, α' will contain the value α would have been assigned if t had not been chosen as a solution. Thus, we explore all combinations of constraints, regardless of order.

5 Results

We implemented our techniques by combining the existing implementations of Helium and Gundry-McBride Unification [5] in the LambdaPi programming language [9], along with a small set of proof-of-concept heuristics. In many cases, our heuristics are able to generate helpful repair hints. We show an example of this in Listing 3: when an equality proof is used in the wrong direction, our heuristics can notify the user of this.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Utrecht Excellence Scholarship.

References

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications*, Luke Ong (Ed.). Lecture Notes in Computer Science, Vol. 6690. Springer Berlin Heidelberg, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

[2] Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863

[3] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (Dec. 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

[4] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde. hhttp://adam.gundry.co.uk/pub/thesis/thesis-2013-12-03.pdf

[5] Adam Gundry and Conor McBride. 2013. A tutorial implementation of dynamic pattern unification. *Unpublished draft* (2013). http://adam.gundry.co.uk/pub/pattern-unify/pattern-unification-2012-07-10.pdf

[6] Jurriaan Hage and Bastiaan Heeren. 2007. *Implementation and Application of Functional Languages: 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Heuristics for Type Error Discovery and Recovery, 199–216. https://doi.org/10.1007/978-3-540-74130-5_12

[7] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. 2003. Constraint based type inferencing in Helium. *Immediate Applications of Constraint Programming (ACP)* (2003), 57.

[8] Bastiaan J Heeren. 2005. Top quality type error messages. *IPA Dissertation Series;* (2005).

[9] Andres Löh, Conor McBride, and Wouter Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Inf.* 102, 2 (April 2010), 177–207. http://dl.acm.org/citation.cfm?id=1883634.1883637