Universiteit Utrecht

Department of Computing Science

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

Improving Error Messages for Dependent Types with Constraint-based Unification

Author:
Joseph Eremondi

Supervisors:
Dr. Wouter Swierstra
Dr. Jurriaan Hage

August 22, 2016

I would like to acknowledge the Utrecht Excellence Scholarship, which made it possible for me to study in Utrecht.

Thank you to my supervisors, Wouter Swierstra and Jurriaan Hage, for their guidance and advice, and for being extremely accommodating as I completed my thesis remotely.

Thank you to Ron Garcia, Josh Dunfield, and all the members of the UBC Programming Languages reading group, for allowing me to attend, and for their valuable feedback on my practice defense.

Thank you to my family, and especially my wife Joanne, for continually providing support and encouragement throughout this process.

Abstract

Dependently typed programming languages provide a powerful tool for proving code correct. However, these languages are complex, and programming in them is often difficult. Certain features, such as pattern matching or inferring values using metavariables, rely on a unification procedure performed during type-checking. When this process fails, the error messages provided are often confusing, and do not accurately indicate which programmer error caused the failure.

For functional languages with simple typing, error messages have been improved by expressing type-checking as a constraint-solving problem, performing global analysis on a constraint-graph. While these attempts have been successful, there are a number of differences between dependent and simple types. Parametric polymorphism is expressed using implicit parameters, rather than polymorphic types. Moreover, since there is no distinction between types and terms, typechecking may require code to be evaluated. This means that the existing techniques for error generation do not directly apply to dependent-types.

This thesis presents background information on dependent-type checking, functional error-message reporting, constraint-based typechecking, and higher-order unification. We frame some of the challenges for adapting existing techniques to more advanced languages.

In an initial attempt to improve error messages, this thesis proposes a dependently-typed system with *replay-graphs*, for combining higher-order unification with type graphs. Additionally, a system of counter-factual constraint-solving is proposed, which finds multiple solutions for various subsets of unification constraints.

The results of implementing these techniques is presented, with examples both of where they produce improved error messages on type-incorrect programs, and where they produce poor messages.

Contents

Chapter 1

Background and Related Work

1.1 Dependently-Typed Languages

We refer the reader to any introductory text for a summary of the untyped, simply typed, and Hindley-Milner varieties of the Lambda Calculus.

1.1.1 Dependent Functions and Products

Dependent types, as we will use them, differ from the Simply-Typed Lambda Calculus in two ways.

First, terms and types are no longer distinct: types can be passed as parameters, and returned by functions. The relation $x : T$ is a relation between two values. For our purposes, we consider a special value Set, where if $x : T$, then $T : \mathsf{Set}$, and $\mathsf{Set} : \mathsf{Set}$. This makes our type system inconsistent, but the measures taken to avoid this are beyond the scope of this thesis.

Secondly, instead of the function type $S \to T'$, we use the type $\Pi\ S\ T$. Here, T must be of the form $\lambda x.\, T'$. We enforce the following typing rules:

$$\frac{\Gamma, x : S \vdash e : T'}{\Gamma \vdash (\lambda x.\, e) : \Pi\ S\ (\lambda x.\, T')} \qquad \frac{\Gamma \vdash f : \Pi\ S\ T \quad \Gamma \vdash x : S}{\Gamma \vdash (f\ x) : (T\ x)}$$

That is, when a function is applied to an argument, the type of the value returned can depend on the argument given.

A similar type, $\Sigma\ S\ T$, can also be used, in place of the pair type (S, T'). Again, T is a function here. We define projection functions fst and snd, with the following typing rules.

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T\ s}{\Gamma \vdash (s, t) : \Sigma\ S\ T} \qquad \frac{\Gamma \vdash p : \Sigma\ S\ T}{\Gamma \vdash \mathsf{fst}\, p : S} \qquad \frac{\Gamma \vdash p : \Sigma\ S\ T}{\Gamma \vdash \mathsf{snd}\, p : T\ (\mathsf{fst}\, p)}$$

These types act as pairs, where the type of the second element can depend on the value of the first.

As a convenience, we will often write $(x : S) \to T'$ to mean $\Pi\ S\ (\lambda x.\, T')$,

The *Curry-Howard Correspondence* provides a connection between logic and programming, where types correspond to theorems and terms correspond to proofs of their types. In this correspondence, simple functions correspond to implication, pairs correspond to the logical *and* operator $\wedge$, and disjoint unions correspond to the *or* operator $\vee$. Adding dependent functions and products extends the correspondence from propositional to first-order logic. The type $\Pi\ S\ (\lambda x.\, T')$ corresponds to $\forall (x : S).\, T'$, and $\Sigma\ S\ (\lambda x.\, T')$ corresponds to $\exists (x : S).\, T'$.

Real-world dependently typed languages implement many features beyond this, but are based on this same principle. Popular modern dependently typed languages include Coq [2], Agda [1], Idris [4], and F* [3]. However, these languages too large and complex for our purposes. Instead, we shall focus on a simpler dependently typed language, LambdaPi [24].

1.1.2 LambdaPi: A Basic Dependently-Typed Language

LambdaPi was introduced to provide a tutorial on type-checking in a dependently-typed language [24]. In addition to dependent functions, it contains a few features, which we highlight here.

Data and Eliminators

It is difficult (though not impossible) to leverage the full power of dependent types using only Π and Σ types. In order to be practical, a few custom *data* types can be added to the language, along with special *eliminator* functions for decomposing them. In LambdaPi, the datatypes $\mathsf{Nat}, \mathsf{Vec}, \mathsf{Eq}, \mathsf{Fin}$ have been added added.

For example, consider natural numbers. We have new values:

$$\begin{gathered}\mathsf{Nat} : \mathsf{Set} \\ \mathsf{Zero} : \mathsf{Nat} \\ \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}\end{gathered}$$

Every number is either zero, or the successor of some other natural number. In place of general recursion, we provide the eliminator:

$$\begin{gathered}\mathsf{natElim} : (m : \mathsf{Nat} \to \mathsf{Set}) \\ \to m\ \mathsf{Zero} \\ \to ((l : \mathsf{Nat}) \to m\ l \to m\ (\mathsf{Succ} l)) \\ \to (k : \mathsf{Nat}) \\ \to m\ k\end{gathered}$$

Semantically, natElim $m\ mz\ ms\ k$ takes a function m mapping numbers to types. It applies the function ms to mz exactly k times, resulting in a value of type $m\ k$. The concept is similar for other types. Fin n is the type of finite natural numbers no larger than n, with a similar eliminator.

We can see the dependently-typed features more clearly with the Vec and its eliminator. Vec $a\ n$ is a type of vectors of length n of elements of type a, whose eliminator corresponds to iterating over the elements. The eliminator vecElim has the following type:

$$
\begin{aligned}
&(a : \mathsf{Set}) \\
&\to (m : (k : \mathsf{Nat}) \to (\mathsf{Vec}\ a\ k) \to \mathsf{Set}) \\
&\to m\ \mathsf{Zero}\ (\mathsf{Nil}\ a) \\
&\to ((l : \mathsf{Nat}) \to (x : a) \to (xs : \mathsf{Vec}\ a\ l) \to (m\ l\ xs) \to (m\ (\mathsf{Succ}\ l)\ (\mathsf{Cons}\ a\ l\ x\ xs))) \\
&\to (k : \mathsf{Nat}) \to (xs : \mathsf{Vec}\ a\ k) \to m\ k\ xs
\end{aligned}
$$

As we can see, we take a function providing a return type for each number k and k-length vector with elements of a. We take a base-case value for an empty vector, and a function that transforms results from a length n vector into those of a length Succ n vector. Finally, we give a number k and a vector of that length.

Eq $a\ x\ y$ is the type of proofs that x and y are equal values of type a, whose eliminator captures the idea that you can substitute x for y when they are equal.

These eliminators allow us to express a large number of programs, without adding pattern matching or recursion to our language.

Bidirectional Type Checking

The typechecking algorithm for LambdaPi is fairly strightforward. Expressions are divided into two categories: *inferred* expressions, whose type we can determine from examining the expressions, and *checked* expressions, that must be annotated with their types. For example, functions must be annotated with their types. Two mutually recursive type-checking functions are defined: an inference procedure, which, given a context and an expression, returns a type, and a checking procedure, which compares an expression with its type, and raises an exception if it is not well typed.

The checking procedure is similar to Hindley-Milner, with a few main differences:

- Since types and values are not distinct, when checking that $e : t$, we may need to evaluate t to normal form.
- All polymorphism is explicit: the Hindley-Milner scheme $\forall a.\, (T\ a)$ is replaced with the ordinary dependent type $(a : \mathsf{Set}) \to T\ a$. For example, $\forall a.\, a \to a$ becomes $(A : \mathsf{Set}) \to A \to A$.

- When checking the application of a function of type $\Pi\ S\ T$ to a value $x : S$, we need to evaluate $(T\ x)$ to normal form to assign a type to the result.
- Lambda values must be annotated with their type. To typecheck $\lambda x.\, e : \Pi\ S\ T$, we create a fresh dummy value L, and check that $e[x/L]$ has type $T\ L$ in the context $\Gamma[L : S]$. This is necessary because the return type of the function depends on its input parameter. While LambdaPi does not feature Σ types, a similar procedure could be used for dependent pairs.

Notably absent from this presentation, when compared to Hindley-Milner, is any concept of unification. Since there is no implicit polymorphism, and functions are always annotated with their types, there is never a time when we assign a value a variable type which is resolved later. As we shall see below, unification will be necessary when pattern matching and implicit polymorphism are introduced.

1.2 Dependently Typed Unification

1.2.1 Spines

It is useful first to have a concept of *spine*-form [7] for representing λ-terms.

Consider the application $f\ x_1\ x_2\ \cdots\ x_n$, where f is a variable. Implicit in this representation is the following parenthesization: $(\cdots((f\ x_1)\ x_2)\cdots\ x_n)$. Notice that, if we were to represent this as a tree, f would be the leftmost-leaf, and we would need to traverse all levels to find it.

In spine form, the above would be represented as $f \cdot [x_1, x_2, \ldots, x_n]$. Here f is referred to as the *head* of the application, and the x_i's as the *spine*. In a tree representation, the head is now the root of the expression.

Since, in unification problems, we often want to unify an application of a function with many arguments against some term, it is useful to think of function applications as being in spine-form. Storing them this way internally increases the performance of unification, allowing easy access to the head. Similarly, this allows us to restrict where unification variables will appear: often we will allow them as spine heads, but not as arguments.

1.2.2 Higher Order Unification

In general, we define the higher-order unification problem for a programming language. Given:

- A set $V = \{\alpha_1, \ldots, \alpha_n\}$ of *metavariables*,
- a set P of problems, of the form $\forall x_1, \ldots, x_k.\, X \equiv Y$, where X and Y are expressions, such that $FV(X), FV(Y) \subseteq V \cup \{x_1, \ldots, x_k\}$, and for each x_i, $x_i \notin V$.

The goal of higher-order unification is to find an assignment of closed terms $e_1, \ldots, e_n$, such that for each problem $\forall x_1, \ldots, x_k.\, X \equiv Y$ in P, we have $X[\alpha_1/e_1] \ldots [\alpha_n/e_n] =_{\beta\eta} Y[\alpha_1/e_1] \ldots [\alpha_n/e_n]$.

That is, we look for a value for each metavariable such that, when we substitute those values for each metavariable, the left hand side and right hand side of each problem are equal.

Higher-order unification is in general undecidable, and for many instances there is no unique solution which generalizes all other solutions. However, Miller [25] identified a sub-problem which is decidable and admits most-general unifiers: the pattern fragment, where metavariables can only be applied to distinct bound variables [6].

1.2.3 Unification in Pattern Matching

Unification for pattern matching is described at length by Norell [27]. To instantiate variables defined in a pattern match on a particular constructor, we must unify the type of the value being matched upon with the return type specified by the particular constructor, and unify the types of the argument constructor with the types of the matched variables. Essentially, the constructor you match upon may refine which types the variables matched upon will have. For example, when you match a value of type Vec a n with Cons x y, we unify Vec a n with Vec b (Succ m).

Unification variables correspond to the inaccessible patterns in the input pattern: patterns whose values or shapes are inferred from the other constructors. In Agda, these are denoted using dot-patterns, which maintain the linearity of the patterns matched upon.

1.2.4 Unification in Implicit Resolution

A second application of unification in dependently-typed languages, on which we focus, arises when we allow the programmer direct access to metavariables. Metavariables can be used in place of expressions, when there is only one possible value the expression can have in order to typecheck correctly.

In Agda and Idris, metavariables are denoted with the underscore character _. We will use this notation in our example programs.

Every metavariable in the program code corresponds to a metavariable used in unification. Type-checking rules are rephrased as constraint-generation rules, a procedure which we will describe in detail in Chapter 3.

For example, consider the following code with labelled metavariables:

```
appendZero : (m : Nat) -> (xs : Vec Nat m) -> Vec Nat (Succ m)
appendZero _1 xs = Cons _2 _3 Zero xs
```

Since we have the type signature
`Cons : (A : Set) -> (n : Nat) -> A -> Vec A n -> Vec A (Succ n)`,
we generate the constraint that $A \equiv Nat$, $\_2 \equiv A$, $\_1 = m$, $\_3 = \_1$, and $n = \_3$.
We can solve these to find that $\_2 \equiv Nat$, and $\_1 \equiv \_3 \equiv m$.

This idea can easily be extended to implicit arguments: a call to a function with an implicit argument is replaced by applying that function to a metavariable. This transformation can be carried out purely syntactically.

1.2.5 Algorithms for Higher-order Unification

There are several algorithms for higher-order unification. Miller identified the *pattern fragment*, the decidable subset of higher-order unification, as well as an algorithm deciding this subproblem[25]. In this form, any time a metavariable is the head of a spine, it is applied only to distinct program variables. So solving for $\alpha\ x\ y \equiv \beta\ x$ is always possible, but solving $\alpha\ x\ x \equiv \beta\ x$ is not, because the arguments to α are not distinct. Likewise, solving $\alpha\ \beta\ x \equiv \beta\ x$ is not always possible, since a metavariable β is an argument to α.

Abel and Pientka present an algorithm that extended Miller's algorithm to a calculus with both Π and Σ types [6]. Gundry and Mcbride further refine the algorithm, and present a tutorial implementation using Haskell [16]. This is expanded upon in Gundry's PhD. Thesis [15].

A more advanced unifcation algorithm, aimed at Coq, is presented by Ziliani and Sozeau [31]. This algorithm supports more advanced dependent type features, such as universe polymorphism. Additionally, it is able to solve a greater number of unification problems through the use of heuristics. Even in a situation where there is no most general unifier, the algorithm will prioritize structural unification. For example, when unifying $f\ [x_1; \ldots x_n]$ with $g\ [y_1; \ldots y_n]$, it will first attempt to unify f with g, and each x_i with y_i, even if f and g are functions whose values we could substitute. This first-order approximation allows for the solutions programmers are likely to expect, when there are many possible solutions.

More recently, a totally different approach to unification in Agda has been proposed [9]. In this system, generating a solution (unifier) also generates a proof of correctness for that solution. The algorithm helps to formalize may previously ad-hoc restrictions which were present in Agda's unification.

Gundry McBride (GM) Unification

As a base, this thesis will use the unification algorithm presented by Gundry and McBride [16, 15]. Because it is written as a tutorial, it was particularly well suited for the scope of this thesis. We refer to it throughout as GM unification.

The algorithm is focused on the manipulation of metacontexts, which store types for metavariables, as well as unification problems. Typing rules are expressed in such a way that expressions are always in normal form. A series of rewrite rules are repeatedly, non-deterministically applied to the metacontext until a solution is found, though the authors describe how this process can be made deterministic.

The algorithm can handle Σ types. Moreover, it is *dynamic*: In the case where some problems are not in the pattern fragment initially, but become

solveable once the substitutions from previous applications are applied, the algorithm can still find a solution.

The algorithm is too complex to describe in its entirety here. We give a description of the features relevant to us in Section 2.3

1.3 Strategies for Improving Error Messages

1.3.1 Guiding Principles

Yang, Michaelson, Trinder and Wells present a *manifesto* for type error reporting [28], with the following main principles:

1. Correctness: errors should be reported if and only if programs are not typeable, and a location is reported only if it contributes to the conflict.
2. Precision: Messages should refer to the smallest relevant portion of the original source code.
3. Succinctness: messages should be short, containing only the most useful information.
4. Amechanicity: artifacts of type inference, such as intermediate variables, should not be presented in the message.
5. Source-based: details of compiler internals or intermediate languages should not leak into error messages.
6. Unbiasedness: when multiple sites contribute to an error, the one which is chosen as an error should be determined by its likelihood of being the cause, not other factors (such as ordering of clauses within the source code).
7. Comprehensiveness: each source site that contributes to the conflict should be reported, and the user should not be required to examine other parts of the source code in order to diagnose the error.

While these are generally desirable goals, sometimes with dependent types it is useful to deviate from them in a few cases. For example, while it is undesirable for details of compiler internals to leak into error messages, there are many cases where unification fails because the problem presented is too sophisticated for the compiler's unification algorithm. Currently, little to no insight is given as to the cause of the unification failure, particularly when a variable's value is unresolved. In situations like this, it may be beneficial to provide some information about the unification algorithm, when it is necessary to guide the programmer to finding a solution.

1.3.2 Error Locations and Slices

As identified in the manifesto, error reporting should reference a minimal but complete set of source locations contributing to the error. A naive approach, such as Algorithm W [11], provides no location for an error, and even more sophisticated approaches only supply the location of the expression that the type-checker was examining when an error was seen.

Haack and Wells introduce the concept of an *error slice* [17]: a set of one or more program points which contribute to the error. An error slice is complete if the relationship between its points guarantees the program will contain a type error, and it is minimal if each contained point is relevant to the error.

The motivation for error slices is that the true location of a fix depends on the desired semantics of a program. Many different changes can cause the program to typecheck, but not all will result in the desired program behavior.

The system they present uses a constraint-based system for type-checking, and attempts to find a minimal set of unsatisfiable constraints. Unification is run multiple times on smaller and smaller sets of constraints, eliminating those which do not contribute to the error until a minimal set is found. Since the number of minimal slices grows exponentially with program size, in practice this procedure is run under a time limit, returning the best result found before time expires.

1.3.3 Counter-Factual Typing

The idea of presenting multiple locations is extended by Chen and Erwig through a technique known as *counter-factual* typing [8]. Like error slicing, it accounts for all possible contributing error locations.

In counter-factual typing, the checker examines atomic expressions and determines what type the expression would need to have in order for the program to typecheck.

The system is based on variational typing and a binary choice-calculus, in which a number of different programs (or, in this case, types) can be represented compactly as a single program, where some sub-expressions contain a choice between two other expressions. A relevant detail is the idea of error-tolerant typing, where ill-typed expressions are explicitly assigned the error type $\perp$, allowing typechecking to continue even when an error is encountered.

Type inference proceeds as in a normal HM system, but when a type ϕ is inferred for an expression e, instead of assigning its type, we assign its type to be a choice between ϕ and a fresh unification variable α. Information about the use of e can then flow into the variable α, giving information about its use, even if it does not match ϕ.

Enumerating all possible choice combinations shows all possible program changes which will lead to well-typedness. Taking the first type in all cases yields the original (possibly failing) typing.

1.3.4 Helium and Constraint-Based Checking

Helium [20] was an alternate Haskell compiler, which aimed to improve the quality of error messages, particularly for beginners. At its core was a type-inference algorithm based on the concept of *constraint-generation*. Where Milner's Algorithm W [26] would unify two variables α, β, Helium emits a constraint of the form $\alpha \equiv \beta$. When expressing that a variable α unifies with an instance of scheme σ, the constraint $\alpha < \sigma$ is emitted.

There are multiple techniques for solving the system of constraints. Algorithm W [26] and Algorithm M [23] were respectively shown to be equivalent to the top-down and bottom-up approaches [19]. Helium could also attempt to reorder constraints using heuristics, then solve them using unification in a way that reduced bias.

For almost completely bias-free analysis, Helium allowed for constraint solution through the construction of a constraint graph [21]. This allowed for constraint solving at the level of the *binding-group* [18]: to solve a constraint set, we can look at more than one constraint at a given time. In this system, type constructors (possibly taking 0 arguments) and unification variables are represented as a node in a graph. Unification edges between nodes α, β are added in this graph whenever the constraint $\alpha \equiv \beta$ is emitted.

For composite types, such as function types, special edges are added from the type constructor (such as $\rightarrow$) to the member types. With composite types, *derived* edges are added to the graph for constraints on their inner types. For example, $t_1 \rightarrow t_2 \equiv t_3 \rightarrow t_4$ adds derived edges $t_1 \equiv t_3$ and $t_2 \equiv t_4$. Adding derived edges may connect nodes, which may trigger the addition of further derived edges.

Typechecking then becomes a simple matter of checking if any two non-equal type-constructors are in the same connected component of the graph, looking only at unification edges.

A comprehensive overview of bias and principles of Haskell error generation can be found in Bastiaan Heeren's PhD. thesis [22]. We will address bias again in Section 1.4.1.

1.3.5 SHErrLoc

Basic HM Inference

An alternate approach to the constraint-based system of Helium is is implemented in the tool known as *SHErrLoc* [29, 5]. Constraints are expressed through an (arbitrary) partial order $\leq$, which are generic enough to describe Hindley Milner typing, as well as security-information flow constraints and dataflow analysis.

In the Hindley-Milner case, a constraint graph is formed with types of expressions as nodes, and two kinds of edges. Edges labeled LEQ denote that the subtype relation must hold between two types (though usually two of these are inserted to stipulate equality). A constructor edge, marked with a constructor

c and index i, when present from u to v, denotes that the type v is the ith value in u, which is built with the given constructor.

Unlike in Helium, where type-checking can be expressed in terms of graph reachability (with derived edges), the form of constructor edges here require that it be expressed in terms of *context-free* reachability, a modified graph problem searching for a path whose labels are in a given context-free langauge. The process of adding new edges to the graph from this procedure is known as *graph saturation*.

In order to accommodate more complex constraints, some constraints have the form $C \vdash C'$. LEQ edges are marked with *hypotheses* C, which can be assumed when trying to verify the constraint C' holds. Once again, the conclusions that can be drawn from a hypothesis can be tested using context-free reachability. For entailments of the form $H \vdash E_1 \leq E_2$, the constraint graph of H, called the hypothesis graph, is constructed to check if it is satisfiable. If it is saturated, then we check if E_1 and E_2 are connected by a LEQ edge in the saturated hypothesis graph.

In order to concisely diagnose which program point is the source of an error, a simple Bayesian heuristic is used, which assumes that a program is mostly correct, and that program points on many unsatisfiable paths are more likely to be a cause of errors than those on many satisfiable paths.

Advanced Type Features

The above system was extended to accommodate more advanced constraints, such as GADTs and Type Families [30]. Here, axioms may be of the form $\forall a.\, C \implies C'$, and type class and type family instances to be expressed using constraints. In order to accommodate non-injective type families, a separate type of node *fun* is introduced in addition to constructor nodes. Typeclasses and type-level functions are encoded into constraints, which are treated as axioms when constructing the constraint graph.

To deal with the more complex constraints, additional nodes and edges are selectively added to the constraint graph, so that all necessary relationships can be inferred though context-free reachability. Specifically, when $LEQ(E, E')$ is in the original graph, then for every occurrence of E within a constructor node, we ensure a node with E' in its place is present, doing the reverse for constructor nodes with E'. The graph is saturated with all edges that can be added this way. Extra measures are taken here to ensure halting, detailed in the paper.

Each LEQ edge in the graph corresponds to an entailment of the form $H \vdash E_1 \leq E_2$. To prove this, a hypothesis graph is formed, with all nodes of the saturated graph. Once the hypothesis graph is formed, it is saturated. During saturation, if an axiom $\forall C.\, I$ has all constraints of C already in the partially saturated hypothesis graph, then I is added, and will contribute to saturation.

To determine if $H \vdash E_1 \leq E_2$ is satisfiable, a search for a substitution θ, in which $\theta(E_1)$ and $\theta(E_2)$ have an LEQ edge in the saturated hypothesis graph, is performed.

While this system supports some type-level computation, it treats the axioms from type-level functions as pre-existing, and does not check type-family instances. It is not powerful enough to deal with dependent types, where the functions applied at type level themselves need to be typechecked.

1.4 The State of Dependently Typed Messages

1.4.1 Causes of Unsatisfactory Error Messages

When a type-error message is given, there is at least one *error*: a section of incorrectly written code, which must be changed in order for the program to typecheck.

The goal of error messages is twofold. First, it tries to identify the section of the code which contains the error, a process known as *error location*.

Secondly, it tries to identify the cause of the error, succinctly informing the programmer as to the cause of the error, provide a hint towards a solution. This process is known as *error classification*.

1.4.2 Left-to-right Bias

Depending on the order a program is traversed in a type-inference algorithm, *bias* can be introduced. Unification variables are generated for various expressions, and may be assigned a type as the algorithm progresses. If another type is assigned to the same variable, that type will be marked as incorrect. The order in which the AST is traversed determines which type is assigned to the expression, and which type is viewed as incorrect.

This kind of bias interferes with the goals of error message generation, since both the assignment of an error location, and the reported error cause, depend on the order in which unifications are performed, as opposed to the actual location or cause.

For example, consider the following Haskell code:

```
let test1 c = if c then "abc" else [True]

  -- Couldn't match expected type 'Char' with actual type 'Bool'
  -- In the expression: True
  -- In the expression: [True]

let test2 c = if c then [True] else "abc"

  -- Couldn't match type 'Char' with 'Bool'
  -- Expected type: [Bool]
  --   Actual type: [Char]
  -- In the expression: "abc"
```

```
  -- In the expression: if c then [True] else "abc"
  -- In an equation for 'test': test c = if c then [True] else "abc"
```

The "expected" versus "actual" errors switch places, depending on which values are provided in the true or false branches.

Since the original publications for Helium [20], GHC error messages have improved: if a type signature is given, this resolves the bias and the return type in the signature is listed as the expected type.

In a simple dependently-typed language, this type of bias is impossible: the requirement that functions are annotated with their types means that the programmer always expresses the intended type of a function.

However, when implicit arguments are introduced and values are inferred, it is once again possible for bias to arise. Consider the following Agda code, with corresponding errors:

```
1  myZipWith
2    : {A B : Set}
3    -> ((A × A) -> B)
4    -> List A
5    -> List A
6    -> List B
7  myZipWith f l1 l2 = Data.List.map f (Data.List.zip l1 l2)
8
9  myVal1 = myZipWith proj₁ (1 :: 2 :: 3 :: []) (true :: false :: [] )
10 -- badFoldFn_tex.agda:9,44-48
11 -- Bool !=< .Data.Nat.Base.ℕ of type Set
12 -- when checking that the expression true has type .Data.Nat.Base.ℕ
13
14 myVal2 = myZipWith proj₁ (true :: false :: [] ) (1 :: 2 :: 3 :: [])
15 -- badFoldFn_tex.agda:14,48-49
16 -- .Data.Nat.Base.ℕ !=< Bool of type Set
17 -- when checking that the expression 1 has type Bool
```

What's more, the reported errors do not change if type annotations are added.

We can see a similar bias in Idris:

```
1  myZipWith
2    : {A, B : Type} -> ((A, A) -> B)
3    -> List A -> List A -> List B
4  myZipWith f l1 l2 = map f (zip l1 l2)
5
6  n : Nat
```

```
n = 22

myVal1 : Nat
myVal1 =
  Prelude.List.length (myZipWith fst [n,n,n] [ True , False])
--at line 11, character 47
--When checking right hand side of myVal1 with expected type Nat
--When checking an application of constructor Prelude.List.::
--Type mismatch between Bool (Type of True) and Nat (Expected type)

myVal2 : Nat
myVal2 =
  Prelude.List.length (myZipWith fst [True, False] [ n ,n,n] )
--at line 19, character 53
--When checking right hand side of myVal2 with expected type Nat
--When checking an application of constructor Prelude.List.::
--Type mismatch between Nat (Type of n) and Bool (Expected type)
```

1.4.3 Amechanicity and Source-Basedness

Another goal we would like with type error messages is *amechanicity*, [28]: we want the messages reported to avoid leaking internal details of the compiler, and to be rooted in the actual source code provided.

For instance, many Agda error messages will refer to intermediate metavariables generated by the unification algorithm.

For example, the following code includes metavariable names, which never appear in its source code, in the error message.

```
myPlus : (ℕ × ℕ) -> ℕ
myPlus (x , y) = x + y


myVal : ℕ
myVal = foldr myPlus 0 [ 1 ]

{-
badPlus.agda:6,15-21
ℕ !=< (_B_8 → _B_8) of type Set
when checking that the expression myPlus has type
ℕ × ℕ → _B_8 → _B_8
-}
```

Even in cases where dependently-typed error messages do not leak compiler internals, they give little useful information about the root cause of the error,

requiring the programmer to understand internal processes such as unification in order to diagnose their type error.

A goal of this project is to produce error messages which are precise, indicating the cause of the problem and potential fixes, without requiring the programmer to fully understand the unification algorithm which solves for metavariables.

1.4.4 Uninformative Messages

Unsolved Metavariables

When a code snippet contains a metavariable which does not have a unique solution, the messages given by Agda contain almost no information.

```
1  natElim
2    : (m : ℕ -> Set)
3    -> (mz : m 0)
4    -> (ms : (l : ℕ )
5    -> (m l) -> m (suc l))
6    -> (k : ℕ)
7    -> (m k)
8  natElim m mz ms zero = mz
9  natElim m mz ms (suc l) = ms l (natElim m mz ms l)
10
11 alwaysZero = natElim _ (zero) (\ x y  -> zero)
12
13 {-
14 _9 : ℕ → Set  [ at NatElim.agda:11,22-23 ]
15 _10 : _9 0  [ at NatElim.agda:11,25-29 ]
16 _11 : _9 0  [ at NatElim.agda:11,25-29 ]
17 _12 : _9 (suc x)  [ at NatElim.agda:11,42-46 ]
18 _13 : _9 (suc x)  [ at NatElim.agda:11,42-46 ]
19 -}
```

Idris reports a similar message, though a type mismatch is reported, as opposed to an unsolved meta error:

```
1  natElim
2    : (m : Nat -> Type)
3    -> (mz : m 0)
4    -> (ms : (l : Nat ) -> (m l) -> m (S l))
5    -> (k : Nat)
6    -> (m k)
7  natElim m mz ms Z = mz
8  natElim m mz ms (S l) = ms l (natElim m mz ms l)
```

```
9
10 alwaysZ : Nat -> Nat
11 alwaysZ = natElim _ (Z) (\ x, y  => Z)
12
13
14 {-
15 NatElim.idr:11:19:
16 When checking right hand side of alwaysZ with expected type
17         Nat -> Nat
18
19 When checking argument ms to function Main.natElim:
20         Type mismatch between
21                 Nat (Type of 0)
22         and
23                 m (S x) (Expected type)
24 Holes: Main.alwaysZ
25 -}
```

Pattern Matching

When a pattern match is invalid, the compiler gives very little indication of the actual cause of the error. For instance, in the following snippet, the error informs us that `x != y`, but never tells us why it is expected that the variables be equal, except perhaps by indicating that `refl` is the error location.

```
1 mysym : (A : Set) -> (x y : A) -> x ≡ y -> y ≡ x
2 mysym A x y refl = ?
3
4 {-
5 BadPatMatch1.agda:2,13-17
6 x != y of type A
7 when checking that the pattern refl has type x ≡ y
8 -}
```

The Idris equivalent faces a similar problem, which is made worse by a poor choice of error location:

```
1 mysym : (A : Type) -> (x : A) -> (y : A) -> x = y -> y = x
2 mysym A x y Refl = Refl
3
4 {-
5 at line 2, character 1
6 When checking left hand side of mysym:
```

```
When checking an application of Main.mysym:
Type mismatch between y = y (Type of Refl) and x = y (Expected type)
Specifically: Type mismatch between y and x
-}
```

1.4.5 Vaguely Located Errors

In the following poorly-typed code, because of the rewrite clause, the programmer is given very little information as to where true source of the error.

```
postulate
  sub2 : (xs : List Bool) → xs ≡ xs ++ []

test2 : (xs : List Bool) → length xs ≡ length (xs ++ [])
test2 xs
  rewrite sub2 xs = refl

{-
Contrib3.agda:5,1-6,25
w != w ++ [] of type List Bool
when checking that the pattern refl has type w ≡ w ++ []
-}
```

Interestingly, similar code does not cause an error in Idris, likely due to different behaviour of the `rewrite` tactic.

```
postulate sub2 : (xs : List Bool) -> xs = xs ++ []

test2 : (xs : List Bool) -> length xs = length (xs ++ [])
test2 xs =
  rewrite (sub2 xs) in Refl

--Compiles without error
```

The following snippet contains almost no information locating the error. Moreover, it gives us very little information as to the *cause* of the error, that is, that the `goodPlus` expects a natural number, and a boolean is given. This cause is muddied by the fact that this constraint is carried through the unification variables, induced by the implicit arguments of `foldr`.

```
goodPlus : ℕ × ℕ -> ℕ -> ℕ
goodPlus (x , y) z = z * (x + y)
```

```
3
4  id : {A : Set} -> A -> A
5  id x = x
6
7  myVal : ℕ
8  myVal =
9    foldr
10           goodPlus
11           0
12           (Data.List.zip (Data.List.map id [] )
13           (Data.List.map id [ true ]) )
14
15 {-
16 badPlusArg.agda:12,6-13,32
17 Bool !=< ℕ of type Set
18 when checking that the expression
19 Data.List.zip (Data.List.map id []) (Data.List.map id [ true ]) has
20 type List (ℕ × ℕ)
21 -}
```

Chapter 2

The Base Language and Type System

Here, we present our staring point: the language and type system which we will augment with type-graphs and counter-factual solving in an attempt to improve error messages.

In short, we add dependent pairs and metavariables to LambdaPi. We omit some of the details, particularly around the datatypes of Eq, Vec and Fin, along with their eliminators.

2.1 The Term and Value Languages

2.1.1 Terms

Our underlying language is a modified version of LambdaPi [24], augmented with existential types, metavariables, and finite numbers (which were in the original implementation, but not the paper). Our source language is described in Figure 2.1:

We allow the programmer to write _ to represent metavariables, assuming that each occurence of _ is replaced with a metavariable β_i through the process described in Section 2.2.1.

We have written eliminators as a separate syntactic construct, in spine form, but for the sake of readability, we will often write $f\ [App\ t; elim_1 \dots elim_n]$ as $(f\ t)\ [elim_1; \dots; elim_k]$, using the usual notation for function application.

We store eliminators as a single list (rather than nested eliminator expressions), so that when we are defining values, we can easily restrict that chains of eliminators are only applied single variables. For example, fst $(f\ x)$ is represented as x [App f; fst].

As is convention, we write $S \to T$ as shorthand for $\Pi\ S\ (\lambda x.\, T)$ where x does not occur in T. For the sake of readability, we will write $\Pi\ S\ (\lambda x.\, T)$ as $(x : S) \to T$ in longer type signatures.

$t ::=$	x	(Variables)
$\mid$	α	(Typechecking metavariables)
$\mid$	β	(Source metavariables)
$\mid$	Set	(The type of types)
$\mid$	$t\ [elim_1; \ldots; elim_k]$	(Eliminating values)
$\mid$	$\Pi\ t_1\ t_2$	(Dependent Function types)
$\mid$	$\Sigma\ t_1\ t_2$	(Dependent pair types)
$\mid$	$\lambda x.\, t$	(Functions)
$\mid$	(t_1, t_2)	(Pairs)
$\mid$	Nat	(Type of natural numbers)
$\mid$	Zero	(Natural number zero)
$\mid$	$\mathsf{Succ}\ t_n$	(Natural number successor)
$\mid$	$\bot$	(Error Value)
$elim ::=$	$\mathsf{App}\ t_{arg}$	(Function Application)
$\mid$	$\mathsf{natElim}\ t_m\ t_z\ t_s$	(Recursion on natural numbers)
$\mid$	fst	(First projection on pairs)
$\mid$	snd	(Second projection on pairs)

Figure 2.1: The Term Language

$$
\begin{array}{rrr}
h ::= & x & \text{(Variable head)} \\
| & \alpha & \text{(Metavariable head)} \\
t ::= & h\ [elim_1; \ldots; elim_k] & \text{Neutral term in spine form}
\end{array}
$$

Figure 2.2: Eliminator forms for values

2.1.2 Values

The set of values is very similar to the set of terms, with the restriction that eliminators (including function application) must only be applied to variables or metavariables. We replace the rule: $t := t\ [elim_1; \ldots; elim_k]$ with the rules in Figure 2.2:

Intuitively, a value is a term on which no evaluation can be performed. As is typical, this forbids values of the form $(\lambda x.\, t)\ s$, but also expressions such as $\mathsf{fst}\ (s, t)$ and $\mathsf{natElim}\ m\ z\ s\ (\mathsf{Succ}\ (\mathsf{Succ}\ \mathsf{Zero}))$.

2.1.3 Semantics

We denote the normal-form of a term t as $[\![t]\!]$, denoting t after all possible evaluation steps have been applied to t or its subterms. Since all computations terminate in this language, it should be clear that the normal form of any expression is a value.

Evaluation occurs when an eliminator is applied to anything other than a variable. We give some example reduction steps in Figure 2.3. We do not give a complete small-step semantics, as all expressions terminate in our language and order of evaluation is not relevant for our purposes. Instead, we give some example reductions, giving a flavour of how the eliminators behave. In particular, structural rules are omitted.

Note that while the semantics are recursively defined, it is not possible to directly write a recursive function.

Function application simply triggers substitution, and the fst *and* snd eliminators are simply pair projections. We can see $\mathsf{natElim}$ represents induction on natural numbers,

2.2 The Type System

2.2.1 Type-checking as Constraint Generation

While Gundry and McBride [15, 16] give an algorithm to solve a set of unification problems, it gives no indication for how to generate unification constraints from a program containing meta-variables. Norell provides an algorithm [27],

$$
\begin{array}{rl}
(\lambda x.\, b)\ t \rightsquigarrow & [x/t]b \\
\\
\mathsf{fst}\ (s, t) \rightsquigarrow & s \\
\mathsf{snd}\ (s, t) \rightsquigarrow & t \\
\\
\mathsf{natElim}\ m\ mz\ ms\ \mathsf{Zero} \rightsquigarrow & mz \\
\mathsf{natElim}\ m\ mz\ ms\ (\mathsf{Succ}\ n) \rightsquigarrow & ms\ n\ (\mathsf{natElim}\ m\ mz\ ms\ n)
\end{array}
$$

Figure 2.3: Example Reduction Steps

ensuring type-safety through *guarded constants*. We instead follow the Gundry-McBride, validating our solutions after-the-fact using definitional equality.

In this section we present the type rules for our language, rephrased to be in a constraint-based form: instead of relying on implicit syntactic pattern-matching in the type rules, we leave types as open variables, expressing separate constraints on them. This allows for the case where the type being checked is a metavariable whose value has not yet been determined.

Our contexts Γ are standard, ordered mappings of variable names to their types.

As in LambdaPi [24], we use bidirectional typing, distinguishing type-inference judgements $:_\uparrow$ from type-checking judgements $:_\downarrow$. We use judgments of the form $e : T \ \&\ C$, meaning that value e has type T when constraint set C is satisfied, with similar judgements for the $:_\uparrow$ and $:_\downarrow$ relations. In a judgement $e : T \ \&\ C$, we require that T be a value, thought e may be any term.

Constraint Language

Our typechecking rules will generate a collection of declarations and constraints. These constraints and declarations will be used as context during the unification procedure. They can take any of the following forms:

$$
\begin{array}{rrl}
C ::= & \alpha : T & \text{(Metavariable declaration)} \\
\mid & t_1 : T_1 \equiv t_2\ T_2 & \text{(Definitional Equality)} \\
\mid & \forall \Gamma.\, t_1 : T_1 \equiv t_2\ T_2 & \text{(Quantified Equality)}
\end{array}
$$

It is no conincidence that these forms are exactly those of context entries in GM unification [15].

For example, when checking that a function has type T, we may generate a constraint $T \equiv \Pi\ \alpha_1\ \alpha_2$, to ensure that the function has a function type.

If $\alpha := \text{fresh}(\Gamma, T)$ & C, where:

$$\begin{aligned} \Gamma &= (x_1 : T_1), (x_2 : T_2), \ldots, (x_k : T_k) \\ \alpha' &= \text{a fresh metavariable} \end{aligned}$$

Then we define our outputs α, C as

$$\begin{aligned} \alpha &= (\alpha'\ x_1\ x_2\ \ldots x_k) \\ C &= (\alpha' : [\![T_1 \to T_2 \ldots \to T_k \to T]\!]) \end{aligned}$$

Figure 2.4: Defining Fresh Typed Metavariables

Metavariable Definition

When solving for an implicit variable inside of an expression, its solution may refer to a variable which was bound in an outer scope. For example, in the (toy) function $\lambda a.\, \lambda x.\, \mathsf{Refl}\ \_\ \_ : (a : \mathsf{Set}) \to (x : a) \to \mathsf{Eq}\ a\ x\ x$, the values to substitute for the underscores are a and x respectively, but these are both bound variables, and we want our solutions to be closed values.

We rectify this by replacing each underscore with a new metavariable, applied to all variables in the current environment. In the above example, we transform the function into $\lambda a.\, \lambda x.\, \mathsf{Refl}\ (\beta_1\ a\ x)(\beta_2\ a\ x)$ with the solutions being $\beta_1 = \lambda a.\, \lambda x.\, a$ and $\beta_2 = \lambda a.\, \lambda x.\, x$, which, when substituted into our original program, give the desired results. Examples such as this further motivate the need for higher-order unification, since nearly every metavariable solution will, in fact, be a lambda-value, abstracted over the context.

We generalize the process in Figure 2.4, where we treat α and C as an output:

Inference Rules

In these rules, when we have a conclusion of the form $\Gamma \vdash e :_\uparrow T$ & C, we consider T and C to be outputs of the inference procedure. Note that the returned T may be a concrete type, or a newly generated metavariable, constrained by members of C.

$$\frac{}{\Gamma \vdash \mathsf{Set} :_\uparrow \mathsf{Set}\ \&\ \emptyset} \qquad \text{(Set of sets)}$$

$$\frac{\alpha := \text{fresh}(\Gamma, \mathsf{Set})\ \&\ C}{\Gamma \vdash \beta :_\uparrow \alpha\ \&\ C} \qquad \text{(Source metavariables)}$$

$$\frac{\Gamma \vdash S :_\downarrow \mathsf{Set}\ \&\ C_1 \qquad \Gamma \vdash s :_\downarrow [\![S]\!]\ \&\ C_2}{\Gamma \vdash (s :: S) :_\uparrow [\![S]\!]\ \&\ C_1 \cup C_2} \qquad \text{(Annotated expressions)}$$

$$\frac{\Gamma \vdash S : \mathsf{Set} \;\&\; C_1 \qquad x \notin \Gamma \qquad \Gamma, x : [\![S]\!] \vdash T\ x :_\downarrow \mathsf{Set} \;\&\; C_2}{\Gamma \vdash \Pi\ S\ T :_\uparrow \mathsf{Set} \;\&\; C_1 \cup C_2}$$
(Universal quantification)

$$\frac{\Gamma \vdash S :_\downarrow \mathsf{Set} \;\&\; C_1 \qquad x \notin \Gamma \qquad \Gamma, x : [\![S]\!] \vdash T\ x :_\downarrow \mathsf{Set} \;\&\; C_2}{\Gamma \vdash \Sigma S T :_\uparrow \mathsf{Set} \;\&\; C_1 \cup C_2}$$
(Existential quantification)

$$\frac{\Gamma \vdash t :_\uparrow T_1 \;\&\; C_1 \qquad \alpha_1 := \mathrm{fresh}(\Gamma, \mathsf{Set}) \;\&\; C_2 \qquad \alpha_2 := \mathrm{fresh}(\Gamma, \alpha_1 \to \mathsf{Set}) \;\&\; C_3}{\Gamma \vdash \mathsf{fst}\, t :_\uparrow \alpha_1 \;\&\; C_1 \cup C_2 \cup C_3 \cup \{\forall \Gamma.\, T_1 : \mathsf{Set} \equiv \Sigma\ \alpha_1\ \alpha_2\}}$$
(Pair first projection)

$$\frac{\Gamma \vdash t :_\uparrow T_1 \;\&\; C_1 \qquad \alpha_1 := \mathrm{fresh}(\Gamma, \mathsf{Set}) \;\&\; C_2 \qquad \alpha_2 := \mathrm{fresh}(\Gamma, \alpha_1 \to \mathsf{Set}) \;\&\; C_3}{\Gamma \vdash \mathsf{snd}\, t :_\uparrow \alpha_2\ (\mathsf{fst}\, t) \;\&\; C_1 \cup C_2 \cup C_3 \cup \{\forall \Gamma.\, T_1 : \mathsf{Set} \equiv \Sigma\ \alpha_1\ \alpha_2\}}$$
(Pair second projection)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :_\uparrow T \;\&\; \emptyset}$$
(Variable lookup)

$$\frac{\begin{array}{cc}\alpha_1 := \mathrm{fresh}(\Gamma, \mathsf{Set}) \;\&\; C_1 & \alpha_2 := \mathrm{fresh}(\Gamma, \alpha_1 \to \mathsf{Set}) \;\&\; C_2 \\ \Gamma \vdash f :_\uparrow T \;\&\; C_3 & \Gamma \vdash t :_\downarrow \alpha_1 \;\&\; C_4\end{array}}{\Gamma \vdash f\ t : (\alpha_2\ [\![t]\!]) \;\&\; C_1 \cup C_2 \cup C_3 \cup C_4 \cup \{\forall \Gamma.\, T : \mathsf{Set} \equiv \Pi\ \alpha_1\ \alpha_2 : \mathsf{Set}\}}$$
(Function application)

$$\frac{}{\Gamma \vdash \mathsf{Nat} :_\uparrow \mathsf{Set} \;\&\; \emptyset}$$
(Natural numbers)

Checking Rules

Notice that in each of the following rules, we never constrain or pattern-match on the form of the type T of the expression we are checking. This allows for the possibility that T is a metavariable, whose value must be determined through constraint solving. Instead, constraints on T are generated.

In order to ensure termination, the rule labeled INF is only applied when no other rules match: otherwise, we could infinitely alternate between that rule and the annotation-checking inference rule. LambdaPi deals with this by internally treating inferred and checked expressions as separate data types.

$$\frac{\Gamma \vdash t :_\uparrow T \;\&\; C_1}{\Gamma \vdash t :_\downarrow T \;\&\; C_1 \cup \{\forall \Gamma.\, T : \mathsf{Set} \equiv T_2 : \mathsf{Set}\}}$$
(INF: Checking an inferred value)

$$\frac{\begin{array}{c}\alpha_1 := \text{fresh}(\Gamma, \mathsf{Set}) \;\&\; C_1 \qquad \alpha_2 := \text{fresh}(\Gamma, \alpha_1 \to \mathsf{Set}) \;\&\; C_2 \qquad x_{free} \notin \Gamma \\ \alpha_3 := \text{fresh}((\Gamma, x_{free} : \alpha_1), \mathsf{Set}) \;\&\; C_3 \qquad \Gamma, x_{free} : \alpha_1 \vdash [x/x_{free}]t :_\downarrow \alpha_3 \;\&\; C_4\end{array}}{\begin{array}{c}\Gamma \vdash (\lambda x.\, t) :_\downarrow T \\ \&\; C_1 \cup C_2 \cup C_3 \cup C_4 \cup \\ \{\forall \Gamma.\, T : \mathsf{Set} \equiv \Pi\ \alpha_1\ \alpha_2, \forall (\Gamma, x : \alpha_1).\, \alpha_3 : \mathsf{Set} \equiv \alpha_2\ x : \mathsf{Set}\}\end{array}}$$
(Lambda Abstraction)

$$\frac{\begin{array}{cc}\alpha_1 := \text{fresh}(\Gamma, \mathsf{Set}) \;\&\; C_1 & \alpha_2 := \text{fresh}(\Gamma, \alpha_1 \to \mathsf{Set}) \;\&\; C_2 \\ s :_\downarrow \alpha_1 \;\&\; C_3 & t :_\downarrow \alpha_2\ s \;\&\; C_4\end{array}}{\Gamma \vdash (s, t) :_\downarrow T \;\&\; C_1 \cup C_2 \cup C_3 \cup C_4 \cup \{\forall \Gamma.\, T : \mathsf{Set} \equiv \Sigma\ \alpha_1\ \alpha_2 : \mathsf{Set}\}}$$
(Forming Existential Pairs)

$$\frac{}{\Gamma \vdash \mathsf{Zero} :_\downarrow T \;\&\; \{\forall \Gamma.\, T : \mathsf{Set} \equiv \mathsf{Nat} : \mathsf{Set}\}}$$
(Natural Zero)

$$\frac{\Gamma \vdash n :_\downarrow \mathsf{Nat} \;\&\; C_1}{\Gamma \vdash \mathsf{Succ}\ n :_\downarrow T \;\&\; C_1\, \{\forall \Gamma.\, T : \mathsf{Set} \equiv \mathsf{Nat} : \mathsf{Set}\}}$$
(Natural Successor)

2.3 Gundry-McBride Unification

The main contribution of this thesis is a modification of the higher-order unification algorithm created by Adam Gundry and Connor McBride [16, 15]. The algorithm itself requires a whole paper to describe. Here, we attempt to highlight the intuition behind it, as well as the parts of it which we will modify later on.

2.3.1 Broad Overview

The unification algorithm keeps a linear (meta)context, the types of metavariables, unification problems and their states, and substitutions from solved problems. At any time, a problem's state can be active, failed, pending on a solution to another problem, or blocked (because it is not in the pattern fragment).

The treatment of blocked problems is what makes the unification algorithm dynamic: even if a problem is not initially in the pattern fragment, it may be in the pattern fragment after other variables are solved and substitutions applied.

Throughout the progress of the algorithm, we will "move" through the context. We keep a cursor marking our location in the context, with the invariant that there are no unapplied substitutions to the left of the cursor, and that problems are always to the right of the declaration of any metavariables they contain.

After collecting constraints and declarations in our typechecking procedure, we begin by placing those constraints to the right of the cursor. The algorithm

moves constraints from right to left, finding solutions to problems as we encounter them. During solving, we may generate more context entries, which we can push to the left or right of the cursor.

There are a number of steps which the algorithm uses to solve problems, which will modify problem state, move the cursor, generate new problems, and so on. When we assign a value for the variable, we move left in the context to its declaration, then move right, applying its new value to problems and trying to solve them, if they are not already solved.

We do not focus on the exact ordering and dependencies of these steps, as it is rather complicated.

2.3.2 Solver Steps

Eta-expansion

When trying to solve $s : S \equiv t : T$, if $S = \Pi\ U\ U'$ and $T = \Pi\ V\ V'$, then we can η-expand the problem into $\forall x.\, s\ x : U'\ x \equiv t\ x : V'\ x$. A similar expansion happens with Σ types. (The actual procedure used is more complicated, relying on a technique of twin-variables to preserve well-typedness, but this is not directly relevant to our work.)

Rigid-rigid Decomposition

We have a *rigid-rigid* matching when unifying two terms, neither of which is a spine with metavariable head.

Rigidly matching two expressions involves checking if their constructors are equal. If they are, equations equalizing their arguments are generated, inserted to the right of the cursor in the context.

For example, $\mathsf{Vec}\ \alpha\ n : \mathsf{Set} \equiv \mathsf{Vec}\ \beta\ n : \mathsf{Set}$ is decomposed into
$\{\alpha : \mathsf{Set} \equiv \beta : \mathsf{Set}, n : \mathsf{Nat} \equiv n : \mathsf{Nat}\}$.

The original GM algorithm performs this step only one level deep, but for our purposes, we assume that we decompose all equations rigidly as deeply down as we can. This will results in simpler constraint graphs being generated later.

In the case where two rigid equations mismatch (i.e. different constructors), an error is thrown, causing the problem to be marked as failed.

Solving a Problem: Inversion and Intersection

A *flex-rigid* equation is one where one side of the equation is a spine-form term with a metavariable head. (Note that the spine may be empty, in the case of $\alpha \equiv t$. Similarly, a *flex-flex* equation is one where both sides of the equation are spine-form terms with a metavariable head.

To solve a flex-rigid equation $\alpha\ [t_1; \ldots; t_n] \equiv t$, we move our cursor left in the context until we find the declaration for α.

If the problem is in the pattern fragment, we generate a solution $\alpha := \lambda x_1 \ldots x_n.\, t$, pushing this to the left, and pushing the substitution to the right. This process is known as *inversion*.

If the problem is not in the pattern fragment, we mark it as blocked, pushing it to the right.

Flex-flex decomposition works much the same way. A special case, however, occurs when our problem is $\alpha[x_1; \ldots; x_n] \equiv \alpha[y_1; \ldots; y_n]$, that is, both equations have the same metavariable head. Here we instead solve by intersection: a fresh metavariable β is generated (with its type pushed left), and we push right the substitution $\alpha := \lambda x_1 \ldots x_n . \beta z_1 \ldots z_k$, where $\{z_1, \ldots, z_k\} = \{x_i \mid x_i = y_i\}$. In this case, we can see that α is defined as a function which ignores all arguments on which x_i and y_i disagree.

Occurs Check

When solving a flex-rigid or flex-flex equation, we must ensure there are no places where α occurs in t, not as an argument to a variable or metavariable. If such an occurrence exists, then there is no solution, and we fail.

For example, there is no finite solution to $\alpha \equiv \Pi\ \alpha\ \beta$, and our occurs check would fail in this case.

Pruning

Before solving a flex-rigid or flex-flex equation, we *prune* the equations. Similar to the occurs check, when solving $\forall \Gamma . \alpha\ [e_1; \ldots, e_k] \equiv t$, we need to ensure all variables in t (which are bound by the $\forall \Gamma$) also occur in the spine $e_1, \ldots, e_k$.

If any are not in the spine, but are arguments to a metavariable within t, we can block, since future solutions may make these disappear. In any other case, there can be no solution.

Verification

In order to ensure safety, after all substitutions have been applied, we verify each equation in the context. Note that the typechecking here is a standard checking algorithm, as in the original LambdaPi, not the constraint-based one used to check whole programs.

We check that, for a metavariable $\alpha : T$ that $T : \mathsf{Set}$, and if the metavariable has a definition $\alpha := t : T$, we check that $t : T$. For equations $\forall \Gamma . s : S \equiv t : T$, we check that $s : S$ and $t : T$, and also that the two sides of the equation are, in fact, identical terms with identical types.

Note that termination does not imply that all metavariables have been defined to values. Rather, it implies that every constraint has been made reflexive. Some metavariables may have no definition, implying that their constraints hold for any value of the proper type. These are reported to the user as errors, since they imply that not enough information is present to infer values omitted in their code.

Chapter 3

Unification for Improved Error Reporting

Here we present the main contribution of this thesis: a modified version of the Gundry-McBride unification algorithm that allows for improved error messages. There are three main part to this:

- An adaptation of Helium's type graphs, which provides a framework for using heuristics to analyse a constraint set simultaneously and identify likely causes of errors and probable fixes.
- Error tolerance, which allows unification to proceed even when errors have been encountered.
- Counter-factual unification, which reduces the bias introduced by the order in which constraints are solved, and provides insight into possible error repairs.

3.1 Type Graphs

In this section, we introduce a graphical representation of unification constraint sets, called *type graphs*. We describe methods for generating them from constraints, as well as for analyzing them in order to diagnose the likely cause of errors.

Type graphs provide an incremental step towards bias-free unification, by using existing type-checking and unification algorithms, referring only to the type graphs for error message generation. They are biased in the sense that the order in which unification problems are solved can still affect the error message output.

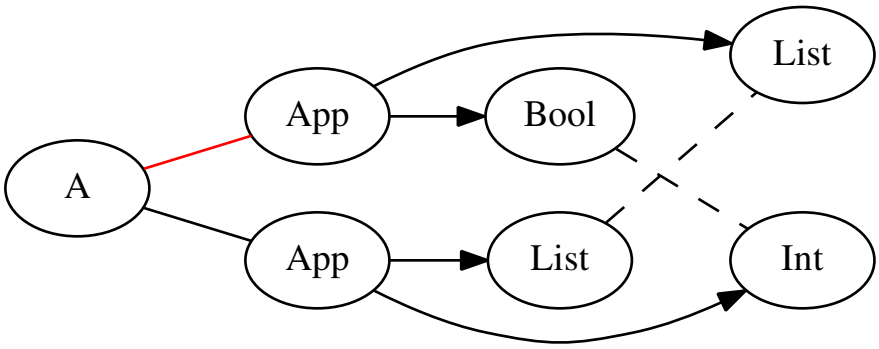

Figure 3.1: A type graph encoding $\{A \equiv List\ Int, A \equiv List\ Bool\}$. The dotted edges are derived, and the red edge represents a possible diagnosis of the error.

3.1.1 Replay Graphs

In Hindley-Milner systems, we can solve a unification problem solely using the type graph. However, this relies heavily on the injective nature of type-constructors: for constructors C, D and terms $s_1, \ldots s_k, t_1, \ldots, t_k$, $C\ s_1\ \cdots\ s_k \equiv D\ t_1\ \cdots\ t_k$ implies that $C = D$ and $s_i \equiv t_i$ for each $1 \leq i \leq k$.

This assumption fails to hold in a higher order setting, because not all functions are injective. For example, it would be extremely foolish to assume that, because $(\lambda x.\, \lambda y.\, \mathsf{Zero})\ \mathsf{Zero}\ \mathsf{Zero} \equiv (\lambda x.\, \lambda y.\, x)\ \mathsf{Zero}\ (\mathsf{Succ}\ \mathsf{Zero})$. that $(\lambda x.\, \lambda y.\, \mathsf{Zero}) \equiv (\lambda x.\, \lambda y.\, x)$, or that $\mathsf{Succ}\ \mathsf{Zero} \equiv \mathsf{Zero}$.

Some steps of the unification algorithm transform problems in order to allow for solving. For example, η-expansion transforms unification on functions into unification on their bodies, and decomposes constraints on existential pairs to constraints on their components.

In order to account for this, we decompose our problem into a number of first-order unification problems by running GM unification and recording the sub-problems and solutions that are generated. When an error is found and a message needs to be generated, the type graph is analyzed to diagnose the cause of the error. Thus, the graph is not used in finding a solution, but instead provides a *replay* of the unification algorithm.

3.1.2 Graph Structure

Our graph structure is very similar to Helium [18, 22], though enriched to allow for a unified type-term language.

Nodes

Nodes represent one of the following:

- A metavariable

- A constructor or constant (such as Nat or Vec)
- A constructor-application, with a left edge to either a constructor or another application, and a right edge to an argument.
- A unique program variable. We treat these the same as constructors, and any edge between two non-equal variables denotes a type error.
- A term node, which represents any higher-order term that cannot be encoded purely using constructors and applications. Lambdas, eliminators, and $\bot$ fall into this category.

Edges

Our edges come in two varieties:

- Directed structure edges, written (u, v, D), where $D \in \{L, R\}$, denoting a left or right edge. These are used to build up complex terms from constructors, variables and metavariables. For example, the edges $(v, v_1, L), (v, \mathsf{Zero}, R), (v_1, \mathsf{Vec}, L), (v_1, \mathsf{Nat}, R)$ denotes that v represents the term Vec Nat Zero.
- Undirected equality edges, written $[u, v]$, denoting that the terms defined by two nodes are definitionally equal. These edges may be explicit, or implied based on equalities of constructor-application nodes.

Graphs and Conversions

As is standard, a graph is a pair (V, E) containing a vertex set V and an edge set E. The graph union $G_1 \cup G_2$ simply denotes $(V_1 \cup V_2, E_1 \cup E_2)$.

There is a conversion function $[\![t]\!]_g = (G, v)$ which produces G, a type graph, and v, the node in G whose value is t. We omit the definition, but it is straightforward, simply copying the term's structure into the node-edge format described above.

The constraint graph for a constraint $C = (s : S \equiv t : T)$, denoted $[\![C]\!]_C$, is defined as $G_1 \cup G_2 \cup (\{v_1, v_2\}, \{[v_1, v_2]\})$, where $[\![s]\!]_g = (G_1, v_1)$ and $[\![t]\!]_g = (G_2, v_2)$.

The connected components of our graph are called *equivalence groups*: any two terms in an equivalence group should be equal after the appropriate substitutions for metavariables are made.

Derived Edges

As in Helium, we use *derived* edges to enforce equality between substructures. It should come as no surprise that the derived edges which are added based on complex term equalities are exactly those which are generated by rigid-rigid decomposition in GM unification: we check that the constructors are equal, and unify the arguments in order.

If there is an undirected edge between two constructor application nodes, we add an undirected edge between their constructors (left children), and an undirected edge between their arguments (right children).

We assume that constructors are syntactic constructs with fixed numbers of arguments, which are not curried. Currying happens by wrapping the constructor expressions in lambdas. For example, when Vec appears in source code, it is actually a name referring to $\lambda x.\, \lambda y.\, \mathsf{Vec}\ x\ y$, not the Vec constructor internal to our language. Because of this, we need not worry about comparing the number of arguments given to constructors, as any partially applied constructors will appear as unapplied lambdas during typechecking.

3.1.3 Constraint Collecting During Solving

Any time our *ambulando* automaton moves right in the context and encounters a constraint C which has not yet been added to our graph, we add $[\![C]\!]_C$ to a global type graph. Simiarly, any time a definition $\alpha := t$ is added to the context, we add an edge $[\alpha, [\![t]\!]_g]$ to our graph.

However, this is not sufficient to express all equalities, since we cannot graphically reason about term nodes. For example, we may have an edge $[\alpha, \lambda x.\, \mathsf{Zero}]$, and another edge $[\alpha\ y, \mathsf{Succ}\ \mathsf{Zero}]$, but cannot deduce the type error that arises from applying alpha. Thus, whenever a value for a metavariable is defined, any term-nodes containing that metavariable are given an edge to their new value when substituting in the metavariable's definition. In the above example, after defining $\alpha := \lambda x.\, \mathsf{Zero}$, we would add an edge $[\alpha\ y, \mathsf{Zero}]$, properly introducing an error path.

In order to facilitate message generation, we also keep a dependency-graph of generated constraints. The sub-constraints of some rigid-rigid unification or η-expansion C are dependent on C, and the definition $\alpha := t$ from solving a flex-rigid or flex-flex problem C is dependent on C. Updates are dependent on their definitions (which in turn are dependent on other problems). Thus, when an edge E is diagnosed as the source of the error, we can trace it back to the original source-code constraint, locating it in the source code. Essentially, all edges which do not come from the initial constraint set are treated as derived edges.

Universal Variables

When adding a constraint of the form $\forall \Gamma.\, C$ to our graph, we replace each variable with a name unique to problem C. These names are shared within all instances of C, but not between problems.

For example, consider the constraints $\forall x.\, \alpha\ x \equiv \beta\ x$, and $\forall x.\, \alpha\ x \equiv \mathsf{Nat}$, that we will refer to as C_1 and C_2 respectively. We will add C_1 with the substitution $[x/x_1]$ and C_2 with $[x/x_2]$, with the variables being distinct, because they are in separate problems.

However, if at some point we define $\alpha := \lambda y.\, y$, when updating C_1 with our new value for α, we generate an edge $x_1 \equiv \beta\ x_1$, and an edge $x_2 \equiv \mathsf{Nat}$

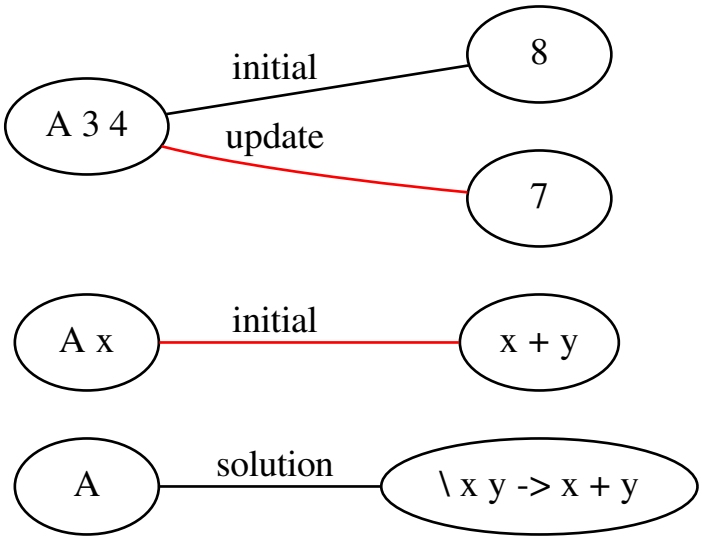

Figure 3.2: The replay graph for $\{A\ x\ y \equiv x + y, A\ 3\ 4\ \equiv 8\}$. The red edges represent a possible diagnosis: an edge on the path from 7 to 8, and the initial edge that induced it.

(which will fail). This way, we can share variables when reasoning within a problem, but will face no naming conflicts when reasoning between problems, since variables are never quantified across more than one problem.

Essentially, each variable is treated as a unique constructor. This is intuitively valid: no two variables are equal, since $\forall x \forall y.\, x \equiv y$ will never hold. Likewise, if a metavariable is defined to be equal to some variable, and also some constructor or value, this will fail, since $\forall x.\, x \equiv t$ is never true.

3.1.4 Constraint Graph Analysis and Heuristics

When the unification solving is complete, we have a final type graph G. A constraint set is unsatisfiable if two unequal constructor or program variable nodes are connected by a path of undirected unification edges. We call any such path an *unsatisfiable path*. (A path from any metavariable to a $\perp$ node also denotes an unsatisfiable constraint set. However, we do not consider these when diagnosing the type graph, opting first to find any mismatch errors we can from the graph.)

A *diagnosis* for an unsatisfiable path is a set of edges which, when removed, causes the two nodes in the path to be in separate connected components. In order to deal with the crowdedness of our graph from the generated constraints, we implicitly remove all edges from constraints dependent on C when we remove C, as well as following Helium's approach and removing derived edges caused by C. This prevents error messages from containing information about

intermediate constraints which are not directly found in the source.

There are, of course, many possible disagnoses for an unsatisfiable path. As in Helium, we use heuristics for deciding which edges to blame. These heuristics are in one of two forms: *avoid* heuristics, which mark certain edges unfit for removal, and *voting* heuritics, which assign votes to each edge. The edge which, after running all heuristics, has the most votes and is not marked unfit, will be removed. This process is repeated until the two conflicting nodes are disconnected.

Unlike Helium, the richer term structure of dependently typed langauges can be used in heuristics. In particular, any eliminators are completely ignored when creating derived edges. However, there is no reason potential heuristics cannot examine eliminators, perform evaluation, or use other higher order techniques.

Edge Information and Inserting Messages

Each edge, when generated, is paired with information about its creation, location in the source code, etc. This can be accessed by the heuristics, allowing them to use this information when diagnosing the probable cause of the error. The edge information is also accessed during error message generation, so that the printed messages can be as detailed as possible.

When a heuristic is examining an edge, it also can modify the stored information for that edge. Through this mechanism, we can suggest fixes or provide other information which was obtained during the global analysis of the type graph, enriching the textual errors presented to the user.

Specific Heuristics

Not all of the Helium heuristics are applicable to dependent types. We outline here the main heuristics from Helium which are well suited to dependent types, as well as some variants of them. In particular, the isomorphism and application heuristics were implemented for our system.

- Participation-ratio heuristic: edges which are on multiple error paths are more likely to be the true cause of an error.
- First-come first-blamed: when all else is equal, ties are broken arbitrarily.
- Trust-factor heuristic: certain constraints, such as those brought about by type annotations, are viewed as more trustworthy, and are therefore avoided when diagnosing errors.
- Permutation heuristic: if swapping two arguments to a function can repair the type graph, the application edge is blamed and the swap is recommended as a fix. We also adapt a version of this heuristic to examine the ordering of type indices, for example to swap $\mathsf{Eq}\ a\ x\ y$ for $\mathsf{Eq}\ a\ y\ x$.
- Tuple heuristic: a specialization of the permutation heuristic to existential pairs.

- Application heuristic: the expected number of arguments for a function is determined from a type graph. Too many are given if an an argument is given to a value without a Π type, and too few when a value non-Π type is expected. Adding or removing arguments is suggested as a possible fix.
- Unifier vertex heruistics: during checking, a fresh type variable often is introduced to make two things equal. When there are conflicting values for this variable, and there is no reason to prefer one over the other, both edges to the unifier are blamed. In a language such as ours, with explicit polymorphism, such a unifier will often be a program metavariable,. For example, consider the element-type argument in `append _ m n xs ys`. If `xs` and `ys` contain we have no way of knowing which is the intended type of the list.

While this set of heuristics forms a solid base for diagnosis, further research is necessary to develop refined heuristics which provide more ideal error messages.

3.2 Error-Tolerant Typing

We now describe modifications we make in order to make GM unification *error-tolerant*: that is, able to continue with unificaiton even after an error has been encountered and it is known there is no solution for a certain metavariable. This allows us to present the most likely error causes, preventing unification from crashing the first time it sees an unsolvable problem.

3.2.1 Rigid-rigid Unification Errors

Any time a rigid-rigid conflict is encountered (i.e. mismatched constructors), GM unification throws an error. Instead, we simply take no action, generating no further subproblems from the given equation.

Because rigid-rigid expansion is defined in the exact same way as derived edges in our type graph, GM unification throws a rigid-rigid error if and only if two disjoint constructors are connected in our final type graph.

3.2.2 An Explicit Error Value

We adapt the concept of the explicit error, presented at least as early as Counter-Factual Typing [8].

We introduce the syntactic form $\bot$, which can be used in place of values, heads, etc. The value of any type-incorrect reduction step is defined to be $\bot$. As an implementation detail, we annotate $\bot$ with a string describing the context of failure, allowing us to recover useful information during error message generation.

Moreover, when we encounter a flex-rigid or flex-flex equation with no solutions, we can assign $\bot$ to the variable whose value we are trying to find, and proceed with unification as if a value had been found.

3.2.3 Changes from GM

We modify GM unification at the following points to increase its error tolerance:

- Rigid-rigid and spine-matching: in these cases, we throw no error, but simply continue unification, since the incorrect match will already be reflected in our type graph.

  Moreover, any value can rigidly match with $\bot$, generating no new problems.

- Occurs check: if we ever have $\alpha\ [x_1; \ldots; x_n] \equiv t$, where α occurs strongly-rigidly (not as a metavariable argument) in t, then any solution for α is infinite. For example, there is no value where $\alpha \equiv \mathsf{Succ}\ \alpha$. In this case, we define $\alpha := \bot$ and continue, giving a special error message to the user when solving is complete.

- Pruning error: there is no solution to $\alpha[x_1; \ldots; x_n] \equiv t$ if there is a variable y, not equal to any x_i, which occurs rigidly in t, that is, not as an argument to a metavariable. If this is the case, we generate the constraint $t \equiv \bot$.

- Typecheck failing: before defining a metavariable $\alpha := t : T$, we check that $t : T$. However, if any sub-expression fails to type-check, instead of failing, we replace it with $\bot$. Thus, we always obtain a result from equalizing and type checking values, though the result be $\bot$, or contain $\bot$ as a sub-expression.

3.3 Correctness of Type Graph and Error-Tolerance

Here, we prove that the success or failure of unification is not changed by our type graph and error-tolerant modifications.

Lemma 3.3.1. *Assume that GM unification throws an error if and only if a constraint set is unsatisfiable. Suppose GM unification succeeds on a constraint-set C. Then all of the following hold:*

1. *There are no inconsistencies in the type graph..*
2. *No unification errors are thrown.*
3. *The solutions for all metavariables do not contain $\bot$ in their definitions.*
4. *The equivalence group of each metavariable in the type graph is equal to the solution generated by GM unification, up to equal variables.*

Proof. 1. Because GM unification encountered no errors, we know that, apart from recording type graph details, our graph performs the exact same steps as GM unification.

Suppose that two constructors C, D are connected by a path in our graph.

Each edge $[u, v]$ on the path was added in one of the following situations:

- $\forall \Gamma.\, u : S \equiv v : T$ was encountered in the context, and $[u, v]$ is not a derived edge.
- $[u, v]$ is a derived edge, and $u \equiv v$ was emitted in GM by the constraint whose edge is inducing $[u, v]$.
- $[u, v]$ was added because v is u after substitution of the solution $\alpha := t$ (or vice versa, switching u and v).

In each case, we can see that the success of GM unification implies that u and v are definitionally equal after all substitutions found by GM unification.

By the transitivity of equality, we can see that C and D are definitionally equal after all substitutions, and since C and D are constructors, which contain no metavariables, $C = D$.

2. Every place the type graph unification throws an error, GM unification also throws an error.

3. Any time our algorithm generates a bottom value, GM unification throws a type error.

4. Suppose GM unification has $\alpha := t : T$ in its context. We show by induction that after n updates to the definition of t, our type graph has t in the equivalence group of α.

   When $\alpha := t$ is initially defined, we add an edge from α to $[\![t]\!]_g$.

   When we update $\alpha := t$ to $\alpha := t'$ by substitution for some metavariables in t, we know by our hypothesis that we have a path from α to t. Our update adds an edge from t to t', so we have a path from α to t.

   Since we already proved our graph is consistent, we know then that we have a solution for α, and that it is correct.

□

Property (4) holds only up to variables because a constraint set such as $\alpha \equiv \beta$ could have solutions $\alpha := \beta$ or $\beta := \alpha$, which are distinct but both correct.

Lemma 3.3.2. *Suppose GM unification fails on a constraint set C. Then at least one of the following holds:*

1. *There is a path between two non-equal constructors in the global type graph.*
2. *A solution for some metavariable contsains $\bot$.*
3. *An error is thrown during unification.*

Proof. If GM unification throws a rigid-rigid error, then a constraint causing the error was added to our graph, so (1) holds.

If GM unification throws a typechecking error, pruning error, or bad occurrence error, then our algorithm produces a solution containing $\bot$ in its place, so (2) holds.

If GM unification throws any other error, then the algorithms performed identical steps until the point the error was thrown, except for recording constraints in our graph. All other failure points in GM unification are present in our type graph algorithm, so our algorithm throws an error as well.

□

These combined give the correctness of our modified algorithm:

Theorem 3.3.3. *Given a constraint set C, GM unification and type graph unification either both fail, or give identical solutions up to equal metavariables.*

3.4 Counter-Factual Unification

Because, in the above sections, type graphs are used only to provide *replays* of unification, they still contain some bias: the order in which we solve constraints can affect the results.

We present *counter-factual unification*: a modification to unification which seeks to reduce bias by exploring solutions arising from subsets of the initial constraint-set. We describe *choice expressions* for compact representation of multiple constraint sets, as well as the modifications to GM unification needed for solving these. We then prove that our modified algorithm halts, and produces equivalent results to GM unification.

3.4.1 Motivation

While there are many potential sources of bias, a particularly problematic one is in substitutions. The order in which constraints are solved in GM unification matters, and when a potential value is found, it is substituted in to all other problems as the "actual" value, and future conflicting values are seen as rigid errors.

The inability of type graphs to handle this case arises from the need to unify arbitrary terms containing metavariables, particularly those containing function applications.

Consider the following unsatisfiable constraint set:
$\{C_1 = \forall x : \mathsf{Set}.\, \alpha\ x : \mathsf{Set} \equiv \mathsf{Vec}\ x\ 0 : \mathsf{Set},$
$C_2 = \forall x : \mathsf{Set}.\, \alpha\ x : \mathsf{Set} \equiv \mathsf{Nat} : \mathsf{Set}\}.$
In GM unification, if we solve C_1 first, we define $\alpha := \lambda x.\, \mathsf{Vec}\ x\ 0$. We substitute the new value in to C_2, and after η-expansion, get $\mathsf{Vec}\ x\ 0 \equiv \mathsf{Nat}$. This update to the value of $\alpha\ x$ is recorded. However, in our type graph, our node for α will only have an edge to $\lambda x.\, \mathsf{Vec}\ x\ 0$. We have a path $\mathsf{Vec}\ x\ 0$ to $\alpha\ x$ to Nat, but the node for α itself occurs on no error paths.

The opposite problem happens if we solve C_2 first, exposing *bias* in our solving procedure. Even with type-graphs, the order in which we solve constraints

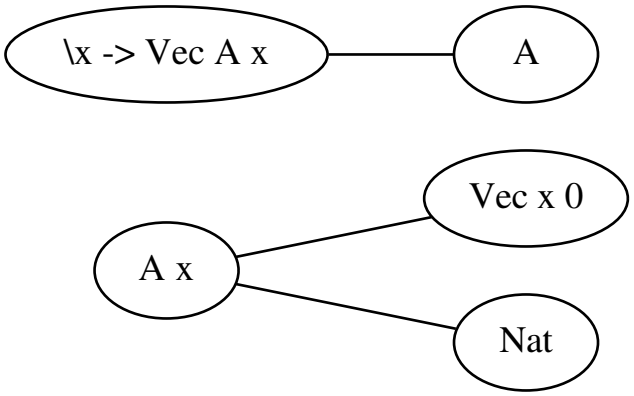

Figure 3.3: The biased type-graph $\{A\ x \equiv \mathsf{Vec}\ x\ 0, A\ x \equiv \mathsf{Nat}\}$. Notice that no expression involving Nat is connected to the node for A.

affects the error results presented. Ideally, we want all conflicting values for α connected to it, so that we can suggest repairs using heuristics.

As a potential solution, suppose that we decomposed the above constraint set into the following: $\{C_1 = \forall x : \mathsf{Set}.\, \alpha_1\ x : \mathsf{Set} \equiv \mathsf{Vec}\ x\ 0 : \mathsf{Set}$,
$C_2 = \forall x : \mathsf{Set}.\, \alpha_2\ x : \mathsf{Set} \equiv \mathsf{Nat} : \mathsf{Set}$,
$C_3 = \alpha_1 : \mathsf{Set} \to \mathsf{Set} \equiv \alpha_2 : \mathsf{Set} \to \mathsf{Set}\}$. If we solve them in that order, we obtain an error path containing both α_1 and α_2.

The following sections formalize the intuition behind this procedure, and details the technique for solving such problems. Specifically, we adapt the ideas from counter-factual typing [8], which in turn is based on the Choice Calculus [13]. These provide a method to counteract bias by examining solutions that arise if we had never given a variable its definition.

3.4.2 A Choice Calculus

We augment our language of values with a form for *named choice expressions:* $t := ... \mid C\langle v_1, v_2\rangle$, where C is some identifier.

Evaluation of expressions involving choices is straightforward: for any eliminators $e_1 \ldots e_n$, we have $C\langle t_1, t_2\rangle\ [e_1 \ldots e_n] = C\langle t_1\ [e_1; \ldots; e_n], t_2\ [e_1; \ldots; e_n]\rangle$.

For each choice identifier C, we define left and right projections C_L, C_R where $C_L\ C\langle t_1, t_2\rangle = t_1$, and $C_R\ C\langle t_1, t_2\rangle = t_2$. We assume that these projections can be applied to arbitrary terms, replacing all sub-terms containing a choice C with the proper variant. Likewise, we assume we can apply these projections to constraints as well, and apply them element-wise to sets of terms or constraints.

Meaning

Suppose that S is some constraint set (though we can just as easily choose a set of terms), and that $C_1, \ldots, C_n$ are the labels of all choices in sub-terms of this

set.

Then, we say that the set represented by this choice set is S_n, where:

- $S_0 = S$
- For $0 < k \leq n$, $S_k = \bigcup_{s \in S_{k-1}}((C_k)_L\ s \cup (C_k)_R\ s)$

That is, a set of choice expressions represents every combination of left and right for each choice label in that expression.

We can see that there are 2^n constraints represented by a term containing n distinct choice labels, assuming each choice has two distinct variants. The calculation is more complicated when choice labels occur more than once, but we can see that a potentially exponential set of values is being represented.

3.4.3 Counter-Factual Solving

The main idea of counter-factual solving is, whenever GM unification generates a subsitution $\alpha := t$, we instead generate $\alpha := \langle t, \alpha_{fresh} \rangle$, so that the solver can also proceed as if no value were ever given to α.

Unlike traditional counter-factual typing [8], we do not implement choice in the type rules. Instead, we use choice when a value for a metavariable is defined.

Defining Metavariables

When we find a solution $\alpha := t$ in a problem P, two actions are taken:

- We immediately apply the substitution $\alpha := t : T$ to problem P in the context.
- We add the substitution $\alpha := C\langle t, \alpha' \rangle : T$ to our context, where C is a fresh choice label and α' is a fresh metavariable. $\alpha' : T$ is added to the left context.

The immediate application is needed to avoid infinite looping: otherwise, any time we defined a variable α in problem P, we would immediately attempt the solution $\alpha' := t$, which would generate a new metavariable, continuing forever.

Freshening: Solving Equations with Choice

An equation with a choice point in fact represents a set of separate equations, which we would like to solve separately, without solutions to one equation affecting the others.

To solve the equation $C\langle s, t \rangle : S \equiv u : V$, we decompose it as follows:

- We split u into $u_L = C_L\ u$ and $u_R = C_R\ u$
- For each metavariable α_i occurring in both u_L and u_R, we declare new metavariables α_i', α_i'', and replace α_i with α_i' and α_i'' in u_L and u_R respectively.

- We iterate left through the context, and before each declaration $\alpha_i : T$, we add $\alpha'_i : T$ and $\alpha''_i : T$.
- We push right the substitution $\alpha_i := C\langle\alpha'_i, \alpha''_i\rangle$
- We push right two new problems, $s : S \equiv u_L : V$, and $t : S \equiv u_R : V$

We refer to the above process as *freshening*, since we are creating instances of the equation with fresh copies of variables. Essentially what we are doing here is decomposing our problem into the two possibilities. However, we do not want any solutions we find when solving $s \equiv u$ to interfere with our solving of $t \equiv u$ or vice versa, which is why we need to generate fresh metavariables.

The splitting of u into u_L and u_R is needed because u itself may contain choice expressions with label C, perhaps from a previous application of this process. We never want to consider the left side of some C choice with the right side of another C choice.

3.4.4 Practical Aspects

When we apply counter-factual solving, we see an exponential explosion in the number of problems and the size of the context, in what is already a potentially slow algorithm. Most of these will be pointless, as we will find equal values for both sides of choice expressions.

In order to facilitate a practical solving algorithm, when we split a constraint involving a choice, as described above, we mark newly generated equation for the right-side of the choice as pending on the failure of the left-hand side. Thus, if the unification problems involving the choice all succeed, we never explore the counter-factual cases.

Likewise, in a real implementation, instead of generating two new variables α', α'' for each metavariable when solving a choice constraint, we can reduce the overall number of variables in the context by not defining α, and replacing its occurrences with $C\langle\alpha, \alpha''\rangle$, never defining or using α'.

3.4.5 Correctness

Lemma 3.4.1. *If GM unification fails on a given problem set S, our counter-factual will also fail on S.*

Proof. Because fresh variables are always generated when solving choice expressions, equations involving the second half of a choice expression will never affect the solutions to the first.

With this in mind, we can see for that every step GM unification performs, a corresponding step is performed by our algorithm. Every time we define a metavariable, the first element in its definition is the GM solution. And every time we solve a choice expression, the first equation generated is identical to what we would encounter in GM, except for possible variable renamings. Thus, whenever GM unification encounters an error, we will encounter a similar one. □

Lemma 3.4.2. *If GM unification succeeds on a given problem set S, our counter-factual system succeeds, where the first projection of the solution to each metavariable present in GM contains the same solution as its GM counterpart.*

Proof. As we discussed in the proof of Lemma 3.4.5, when we ignore the counter-factual parts of equations and definitions, an identical process to GM unification is performed by our algorithm, with some possible variable renamings. Since the newly generated equations and variables do not affect the factual case, any solutions will be identical.

Moreover, if the constraint set is consistent (because GM unification solved it), we can show that no counter-factual equations will fail. This is because, when we make a new variable α' solving α, any constraint involving α' were derived from constraints that originally involved α. A similar fact can be shown when we freshen α to α', α''. So for any newly generated counter-factual variable, there exists some factual variable for which the constraints placed on the counter-factual variable are no more than those on the factual one. Since we know the constraint-set is consistent with regards to factual variables, it must also be for counter-factual ones. □

In addition to this, we cannot neglect the fact that we are expanding the context and generating new equations. Here we prove that our process will still halt whenever GM unification halts.

Lemma 3.4.3. *Counter-facutal unification halts on any initial constraint set.*

Proof. Intuition for the halting proof of GM unification is given by Gundry [15]. We show why our process of generating new problems and choice definitions is well founded.

We examine the behaviour of the algorithm by considering the set of problem sets represented by all choices, which we call the *true* problem sets.

Since the algorithm behaves as GM at first, we know we will eventually define at least one metavariable, or halt before doing so. We show by induction that, if there are n occurrences of undefined metavariables in the true problem set for the context, before making a definition $\alpha := C\langle t, \alpha' \rangle$, the algorithm will use only a finite number of steps after applying the definition.

If there is 1 variable occurrence in the true sets S, then that variable is in the problem generating the definition. We substitute t for α immediately, and since there are no other occurrences, we never use α'. So all variables are solved and the algorithm halts.

Suppose there are $n > 1$ variables in the true problem sets S before our definition. After our definition, we obtain a new set S', by substituting the definition of α. Each problem in a set in S' will be one from a set in S, with either t or α' substituted in. So our true problem set can be partitioned into S', the problems for the factual case of α, and S'', for the counter-factual case.

We showed in the above lemmas that the process of solving these two partitions is completely independent. So if we solve them separately and halt, then solving them together will also halt.

We can apply our induction hypothesis to each of the two parts. Since we removed one variable from the problem that defined α (and did not substitute in α'), each half has less than n variable occurrences, so our hypothesis holds, and each half can be solved in finite time. In particular, this holds because the process of freshening metavariables does not add any variables to the true problem sets, it simply splits a choice problem into two non-choice problems. Other operations from GM unification may increase the occurrences in the context, but since we know GM unification halts, these operations still move us closer to halting.

□

3.4.6 Counter-Factual Type-Graphs

The use of counter-factual typing helps generate potential solutions for ill-typed metavariables. However, we still wish to take advantage of the information available to us from type graphs, and the heuristics for diagnosing error locations.

Our solution is very simple: when we generate a definition $\alpha := C\langle t, \alpha' \rangle$, we add a constraint-graph edge between α and α'. Similarly, when we freshen a variable α to α', α'', we add edges from α to each of α', α''. This expresses the constraint that, in a well typed program, the counter-factual case will produce a result unifyable with the factual case.

Repair Heuristics

We can easily use counter-factual solving in a repair heuristic. Whenever we define $\alpha := C\langle t, \alpha' \rangle$, we have an edge from α to t, and one from α to α'. If we remove the edge from α to t (as well as its dependent edges), and the new graph has fewer inconsistencies, then we can suggest the value of α' as a likely fix for the value of α.

Chapter 4

Results and Discussion

In order to evaluate our techniques, we combined the implementations of Helium, LambdaPi, and GM Unification. Our implementation can be found on GitHub [12]. A few of the Helium heuristics, such as the application and permutation heuristics, were transferred to our system, which allowed for the hint suggestions showcased below.

4.1 Comparison of Error Messages

Here, we present some simple programs containing type errors, with roughly equivalent versions presented in Agda, Idris, and our variant of LambdaPi. We present the different error messages reported for each.

An issue with our language is that several errors are often identified, where only one should be. For the sake of these examples, we simplify the messages presented to a single error, in order to highligh the repair heuristics of our system. We discuss the problems with the full errors in Section 4.2.

4.1.1 Too Many Arguments

Our compiler has modest improvements in thise case. While the Idris and Agda messages aren't particularly terrible, we are able to use the type graph to infer the expected number of arguments for the given goal type.

Unfortunately, due to limitations of our version of the application heuristic, metavariables are left in the reported expected type.

Agda

```
1 {-#  OPTIONS --type-in-type #-}
2 module TooManyArgs where
3
4 open import AgdaPrelude
```

```
5
6  myFun : (a : Set) -> a -> a -> a
7  myFun a x y = x
8
9  myApp = myFun _ Zero Zero Zero Zero
10
11 -- TooManyArgs.agda:9,9-26
12 -- Nat should be a function type, but it isn't
13 -- when checking that Zero Zero are valid arguments to a function of
14 -- type Nat
```

Idris

```
1  module TooManyArgs where
2
3  open import AgdaPrelude
4
5  myFun : (a : Set) -> a -> a -> a
6  myFun a x y = x
7
8  myApp = myFun _ Zero Zero Zero Zero
9
10 -- TooManyArgs.idr:8:15:
11 -- When checking right hand side of myApp with expected type
12 --         Nat
13 -- When checking an application of function TooManyArgs.myFun:
14 --         Type mismatch between
15 --                 Nat (Type of Zero)
16 --         and
17 --                 _ -> _ (Is Zero applied to too many arguments?)
```

Our Compiler

```
1  let myFun = (\ a x y -> x) :: forall (a :: *) . a -> a -> a
2
3  let myApp = myFun _ 0 1 2
4
5
6  --   TooManyArgs.lp: 3,13 Mismatch in type of
7  --   myFun _ (0 :: Nat) (1 :: Nat) (2 :: Nat)
8  --     ?A_3_25__3 -> (?A_3_25__18)  =/= Nat
9  --     HINT: Function expected at most 3 arguments, but you gave 4
```

The reported message does not change with counter-factual solving enabled.

4.1.2 Too Few Arguments

In the case where too few arguments are given, our repair heuristics are able to suggest both the types and positions of arguments which must be added in order to create a well-typed function call. Agda and Idris, however, only inform the user that the actual type of the expression is a function type, describing its mismatch with the expected result type.

Agda

```
{-#  OPTIONS --type-in-type #-}
module TooFewArgs where

open import AgdaPrelude

myFun : (a : Set) -> a -> a -> a
myFun a x y = x

myApp : Nat
myApp = myFun _ Zero

-- TooFewArgs.agda:10,9-21
-- Nat -> Nat !=< Nat of type Set
-- when checking that the expression myFun _ Zero has type Nat
```

Idris

```
module TooFewArgs

import IdrisPrelude

myFun : (a : Type) -> a -> a -> a
myFun a x y = x

myApp : Nat
myApp = myFun _ Zero

-- TooFewArgs.idr:9:7:
-- When checking right hand side of myApp with expected type
--         Nat
-- Type mismatch between
--         a -> a (Type of myFun a _)
```

```
-- and
--          Nat (Expected type)
```

Our Compiler

```
let myFun = (\ x y -> x) :: Nat -> Nat -> Nat

let myApp = (myFun 0) :: Nat

--   TooFewArgs.lp: 3,14 Mismatch in type of result of application
--   myFun (0 :: Nat)
--     Nat =/= Nat -> Nat
--     HINT: Function expected 2 arguments, but you gave 1.
--     Try ("myFun") (Zero) (x2)
--       where
--         x2 :: Nat
```

The reported message does not change with counter-factual solving enabled.

4.1.3 Arguments in the Wrong Order

Similarly to the case with too few arguments, our repair heuristics can suggest a permutation of the given arguments which is likely to resolve the error. Agda and Idris, instead, report only the first ill-typed argument as an error, providing no repair suggestions.

Agda

```
{-#  OPTIONS --type-in-type #-}
module ArgsWrongOrder where

open import AgdaPrelude

myFun : (a : Set) -> a -> Nat -> Nat
myFun _ x y = y

myApp = myFun _ Zero (Nil Nat)

-- ArgsWrongOrder.agda:9,23-30
-- Vec Nat Zero !=< Nat of type Set
-- when checking that the expression Nil Nat has type Nat
```

Idris

```
{-#  OPTIONS --type-in-type #-}
module ArgsWrongOrder where

open import AgdaPrelude

myFun : (a : Set) -> a -> Nat -> Nat
myFun _ x y = y

myApp = myFun _ Zero (Nil Nat)

-- ArgsWrongOrder.idr:12:15:
-- When checking right hand side of myApp with expected type
--         Nat
-- When checking an application of function ArgsWrongOrder.myFun:
--         Type mismatch between
--                 Vec a Zero (Type of [])
--         and
--                 Nat (Expected type)
```

Our Compiler

```
let myFun = (\ _ x y -> y) :: forall (a :: *) . a -> Nat -> Nat

let myApp = (myFun _ 0 (Nil Nat))

--  ArgsWrongOrder.lp: 3,14 Mismatch in type of
--  myFun _ (0 :: Nat) (Nil Nat)
--    Nat =/= Vec Nat Zero
--    HINT: Function arguments in the wrong order.
--    Try ("myFun") _ (Nil Nat) (Zero)
```

When counter-factual solving is enabled, our heuristics fail in this case:

```
--   ArgsWrongOrder.lp: 3,25 Mismatch in type of result of application
--   Nil Nat
--     Vec Nat Zero =/= Nat
```

4.1.4 Dependent Type Error

Here we show the error from a classic, dependent-type specific error: reversing the indices of an Eq value. The programmer uses Eq Nat (Succ x) ($plus$ (Succ x) 0)

in the type signature, where the type should in fact be Eq Nat ($plus$ (Succ x) 0) (Succ x). This example can be fixed by reversing the type signature, or using a proof of equality's symmetry.

Here, instead of an application heuristic, we are able to apply an isomorphism heuristic, seeing that the two conflicting types are isomorphic to one another, and prompting the user to rearrange to repair them. However, we can only do this when counter-factual solving is enabled.

Agda

```
module BadRefl where

open import AgdaPrelude

plus =
  natElim
    ( \ _ -> Nat -> Nat )            -- motive
    ( \ n -> n )                     -- case for Zero
    ( \ p rec n -> Succ (rec n) )    -- case for Succ


postulate pNPlus0isN : (n : Nat) -> Eq Nat (plus n Zero) n



succPlus : (n : Nat) -> Eq Nat  (Succ n) (plus (Succ n) Zero)
succPlus =
  (\n -> pNPlus0isN (Succ n))

-- BadRefl.agda:18,10-29
-- natElim
--    (\ _ → Nat -> Nat)
--    (\ n_1 -> n_1)
--    (\ p rec n_2 → Succ (rec n_1))
--    n Zero
-- != n of type Nat
-- when checking that the expression pNPlus0isN (Succ n) has type
-- Eq Nat (Succ n) (plus (Succ n) Zero)
```

Idris

```
module BadRefl

import IdrisPrelude
```

```
4
5  plus : Nat -> Nat -> Nat
6  plus =
7    natElim
8      ( \ _ => (Nat -> Nat) )           -- motive
9      ( \ n => n )                     -- case for Zero
10     ( \ p, rec, n => Succ (rec n) )   -- case for Succ
11
12
13 postulate pNPlus0isN : (n : Nat) -> Eq Nat (plus n Zero) n
14
15
16
17 succPlus : (n : Nat) -> Eq Nat  (Succ n) (plus (Succ n) Zero)
18 succPlus =
19   (\n => pNPlus0isN (Succ n))
20
21 BadRefl.idr:19:21:
22 When checking right hand side of succPlus with expected type
23         (n : Nat) -> Eq Nat (Succ n) (plus (Succ n) Zero)
24
25 -- When checking argument n to BadRefl.pNPlus0isN:
26 --         Type mismatch between
27 --             Succ (natElim (\underscore => Nat -> Nat)
28 --                           (\n1 => n1)
29 --                           (\p => \rec => \n5 => Succ (rec n5))
30 --                           n
31 --                           Zero) (Inferred value)
32 --         and
33 --             Succ n (Given value)
34 --
35 --         Specifically:
36 --                 Type mismatch between
37 --                     natElim (\underscore => Nat -> Nat)
38 --                             (\n1 => n1)
39 --                             (\p => \rec => \n5 => Succ (rec n5))
40 --                             n
41 --                             Zero
42 --                 and
43 --                     n
```

Our Compiler

Unfortunately in this case, our heuristics falsely identify this as a wrong-arguments error, suggesting a fix that is, in fact, identical to the original program.

```
-- addition of natural numbers
let plus =
  natElim
    ( \ _ -> Nat -> Nat )            -- motive
    ( \ n -> n )                     -- case for Zero
    ( \ p rec n -> Succ (rec n) )    -- case for Succ

assume 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)


-- ERROR: BadReflPost.lp: 12,10 Cannot solve the following constraints:
--
--   BadReflPost.lp: 12,10 Mismatch in type of result of application
--   pNPlus0isN (Succ [Free_ (Local 0)])
--     Eq Nat ((Succ (x)))
--       (Succ
--         ((natElim
--           (\ arg . Nat -> Nat)
--           (\ arg . arg)
--           (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--     =/=
--      Eq Nat
--       (Succ
--         ((natElim
--           (\ arg . Nat -> Nat)
--           (\ arg . arg)
--           (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--       ((Succ (x)))
--     HINT: Function arguments in the wrong order.
--           Try ("pNPlus0isN") (Succ (x))
```

Our Compiler: Counter-Factual

When we enable counter-factual solving, one of the messages contains the following repair hint:

```
--   BadReflPost.lp: 12,10 Mismatch in type of result of application
--   pNPlus0isN (Succ [Free_ (Local 0)])
--     Eq Nat ((Succ (x)))
```

```
--        (Succ
--          ((natElim
--            (\ arg . Nat -> Nat)
--            (\ arg . arg)
--            (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--      =/=
--       Eq Nat
--        (Succ
--          ((natElim
--            (\ arg . Nat -> Nat)
--            (\ arg . arg)
--            (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--        ((Succ (x)))
--      HINT: Rearrange arguments to match
--      Eq Nat ((Succ (x)))
--        (Succ
--          ((natElim
--            (\ arg . Nat -> Nat)
--            (\ arg . arg)
--            (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--      to
--       Eq Nat
--        (Succ
--          ((natElim
--            (\ arg . Nat -> Nat)
--            (\ arg . arg)
--            (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero))
--        ((Succ (x)))
```

As ugly as this is, if we substitute for definitions of `plus`, we see a more helpful hint:

```
--      HINT: Rearrange arguments to match
--        Eq Nat (Succ x) (plus (Succ x) 0)
--      to
--        Eq Nat (plus (Succ x) 0) (Succ x)
```

4.2 Limitations

4.2.1 Error Redundancies

Because each inconsistency in the type graph is marked as an error, what is conceptually one error is sometimes seen as two or more.

For example, in Subsection 4.1.2, our compiler reports the given error message twice. Each message corresponds to a separate path from the type of *myApp* to $\mathsf{Nat} \to \mathsf{Nat}$ in the graph, when the graph already has a path from that node to Nat.

Further study is needed to determine whether this is simply a flaw of our current approach or implementation, or whether this is inherent to replay graphs. In any case, a more complete implementation could use heuristics to determine the ideal message to present to the user, and to filter out duplicate errors.

Counter-Factual Redundancies

The redundancies identified above are made worse with counter-factual solving. Because counter-factual solving performs every step of normal GM unification in the factual case, many errors will cause multiple graph inconsistencies: one for the inconsistency GM unification would cause, and one for disagreement between the factual and counter-factual case.

Sub-Structure Redundancies

There are also some redundancies present with the building up of sub-structures.

For example, in the Eq example above, our compiler actually reports a conflict between Eq (*plus* (Succ x) 0) (Succ x), Eq (Succ x) (*plus* (Succ x) 0), Eq (Succ x) (Succ x), and Eq (*plus* (Succ x) 0) (*plus* (Succ x) 0). This is because internally, it has Eq α β, where α and β are each metavariables with (Succ x) and (*plus* (Succ x) 0) as conflicting values.

4.2.2 Derived Edges in Repair Heuristics

Our compiler uses replay graphs to generate derived edges from initial constraints. However, many heuristics involve removing and adding edges in the graph.

While removing the derived edges from an initial edge is simple, generating derived edges when new edges are added is not. Re-running the unification algorithm would likely be too costly for practical purposes.

Because of this, some of our heuristics generate "false positives", places where a suggestion is given that will not actually resolve the type errors, because the derived edges cause inconsistencies that are not immediately apparent in the graph. This is why the example without counter-factual typing fails in Section 4.1.4.

Controlling Evaluation

As we saw Section 4.1.4, the error messages our compiler generates are often long and unwieldly. This is because Gundry and McBride's implementation, and hence ours, is extremely strict in its evaluations, opting to always represent values in normal form. As a result, named functions are prematurely substituted in for their definitions.

For example, the following function occurs repeatedly.

```
((natElim
            (\ arg . Nat -> Nat)
            (\ arg . arg)
            (\ arg arg1 arg2 . Succ ((arg1 (arg2))))) x Zero)
```

This is simply the definition of *plus*. Controlling evaluation more carefully could prevent *plus* from being expanded, resulting in a much more readable error message.

4.3 Future Work

It goes without saying that future work can include investigating and improving on the above limitations. However, we highlight a few more potential research topics:

4.3.1 Improved Reporting Heuristics

We have presented a framework, which allows for the analysis of type-errors using heuristics on type graphs. We have adapted a few Helium heuristics to our framework. However, there is much room for development of heuristics specifically targeted at dependent types.

Ideally, in the future, a large collection of programs written by actual beginners to dependently-typed languages would be collected, as in Helium, with ideal error locations identified by hand. Heuristics could then be tuned on these examples, to identify the most helpful constraint in most or all cases.

Likewise, future work could examine the Baysean heuristics from SHErrLoc [30], applying them to the our type-graph format, or creating a form of replay graphs using the SHErrLoc graph format.

4.3.2 Performance Considerations

The unification algorithm which we adapted was, according to the authors, not tuned for performance, with much needless iteration through the context being performed. Future work on improving that algorithm would certainly carry over to ours, yielding speed improvements.

Similarly, the use of counter-factual solving adds exponential slowdown to an already slow algorithm. While we have suggested some techniques for keeping the number of counter-factual cases low, a more sophisticated algorithm could yield performance improvements.

Switch combinators were developed for Helium [22] to address the computational costs associated with type-graph analysis. Possible future work could examine how these would need to be changed to be adapted to our system.

The lack of recursion in our language keeps binding-groups small: we only analyze one function at a time. Languages such as Idris require mutually recursive functions to be declared specially [10], though Agda recently lifted this requirement [14]. In any case, the requirement that all functions be terminating will likely force users to keep track of their recursion carefully, possibly keeping binding groups small.

4.3.3 Alternate Constraint Solvers

While our algorithm is based on GM unification, there are several different higher order unification algorithms [6, 31, 9]. GM unification was ideal for the scope of a Masters thesis, but other algorithms could be more performant, and able to solve a wider set of problems. The concepts we introduce, such as replay-graphs and counter-factual solving, could likely be applied to more sophisticated constraint solvers.

4.4 Conclusion

In this thesis, we presented previously known techniques for error message improvement in functional languages. We identified the difficulties with adapting these techniques to dependently-typed languages.

With replay graphs and counter-factual solving, we have provided an attempt at improving error messages in dependently-typed languages. These techniques were implemented in a simple language, and initial evaluation showed that helpful hints could be generated for simple examples.

The implementation is rough and preliminary, and our approach taken has limitations, providing worse messages in some cases. However, we have shown that it is possible to use more advanced error techniques with dependent types.

This thesis has identified possible approaches, as well as challenges, for dependent type error generation. It is our hope that it will provide a solid base on which future improvements can be made.

Bibliography

[1] The Agda programming language. http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed: 2016-01-27.

[2] The Coq theorem prover. https://coq.inria.fr/. Accessed: 2016-01-27.

[3] The F* project. http://research.microsoft.com/en-us/projects/fstar/. Accessed: 2016-01-27.

[4] Idris: A language with dependent types. http://www.idris-lang.org/. Accessed: 2016-01-27.

[5] SHErrLoc project. http://www.cs.cornell.edu/projects/SHErrLoc/. Accessed: 2015-12-16.

[6] Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In Luke Ong, editor, *Typed Lambda Calculi and Applications*, volume 6690 of *Lecture Notes in Computer Science*, pages 10–26. Springer Berlin Heidelberg, 2011.

[7] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[8] Sheng Chen and Martin 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, New York, NY, USA, 2014. ACM.

[9] Jesper Cockx, Dominique Devriese, and Frank Piessens. Unifiers as equivalences: Proof-relevant unification of dependently typed data. *To appear at ICFP 2016*, 2016. https://people.cs.kuleuven.be/~jesper.cockx/unifiers-as-equivalences/draft.pdf,.

[10] The Idris Community. Documentation for the idris language. http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html#dependent-types, 2015.

[11] Luis Damas and Robin 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, New York, NY, USA, 1982. ACM.

[12] Joseph Eremondi. Github repository: lambda-pi-constraint, tag thesis-final. `https://github.com/JoeyEremondi/lambda-pi-constraint`, 2016.

[13] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011.

[14] Ulf Norell et. al. Agda documentation. `http://agda.readthedocs.io/en/latest/language/mutual-recursion.html`, 2016.

[15] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[16] Adam Gundry and Conor McBride. A tutorial implementation of dynamic pattern unification. *Unpublished draft*, 2013.

[17] Christian Haack and J.B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1–3):189 – 224, 2004. 12th European Symposium on Programming (ESOP 2003).

[18] Jurriaan Hage and Bastiaan Heeren. *Implementation and Application of Functional Languages: 18th International Symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, chapter Heuristics for Type Error Discovery and Recovery, pages 199–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[19] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183, April 2009.

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

[21] Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical report, 2001.

[22] Bastiaan J Heeren. Top quality type error messages. *IPA Dissertation Series;*, 2005.

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

[24] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inf.*, 102(2):177–207, April 2010.

[25] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, October 1992.

[26] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[27] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[28] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *IFL '00*, pages 71–86, RWTH Aachen, September 2000. Springer Verlag.

[29] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 569–581, New York, NY, USA, 2014. ACM.

[30] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 12–21, New York, NY, USA, 2015. ACM.

[31] Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 179–191, New York, NY, USA, 2015. ACM.