Practical Dependent Type Checking Using Twin Types

Víctor López Juan
Chalmers University of Technology
Gothenburg, Sweden
victor@lopezjuan.com

Nils Anders Danielsson
University of Gothenburg
Gothenburg, Sweden
nad@cse.gu.se

Abstract

People writing proofs or programs in dependently typed languages can omit some function arguments in order to decrease the code size and improve readability. Type checking such a program involves filling in each of these implicit arguments in a type-correct way. This is typically done using some form of unification.

One approach to unification, taken by Agda, involves sometimes starting to unify terms before their types are known to be equal: in some cases one can make progress on unifying the terms, and then use information gleaned in this way to unify the types. This flexibility allows Agda to solve implicit arguments that are not found by several other systems. However, Agda's implementation is buggy: sometimes the solutions chosen are ill-typed, which can cause the type checker to crash.

Gundry and McBride's twin variable technique also allows starting to unify terms before their types are known to be equal, and furthermore this technique is accompanied by correctness proofs. However, so far this technique has not been tested in practice as part of a full type checker.

We have reformulated Gundry and McBride's technique without twin variables, using only twin types, with the aim of making the technique easier to implement in existing type checkers (in particular Agda). We have also introduced a type-agnostic syntactic equality rule that seems to be useful in practice. The reformulated technique has been tested in a type checker for a tiny variant of Agda. This type checker handles some challenging examples that Coq, Idris, Lean and Matita cannot handle, and does so in time and space comparable to that used by Agda. This suggests that the reformulated technique is usable in practice.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

TyDe '20, August 23, 2020, Jersey City, NJ
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

CCS Concepts • **Theory of computation** → Type theory.

Keywords type checking, unification, dependent types

ACM Reference Format:
Víctor López Juan and Nils Anders Danielsson. 2020. Practical Dependent Type Checking Using Twin Types. In *TyDe '20: ACM Workshop on Type-Driven Development, August 23, 2020, Jersey City, NJ*. ACM, New York, NY, USA, 13 pages. https://doi.org/XX.XXXX/XXXXXXX.XXXXXXX

1 Introduction

Dependent types are the basis of programming languages/proof assistants such as Agda, Coq, Idris, Lean and Matita. Such languages typically allow users to omit pieces of code, like certain function arguments: this can make code easier to read and write. Type checking such a program includes finding type-correct terms for the omitted pieces of code, and this tends to involve solving higher-order unification problems, which in general is undecidable [11]. Furthermore, even if one can find all solutions there might not be a most general one. If a solution which is not a most general one is picked, then this could potentially mean that the program does not mean what the programmer intended.

Some systems, for instance Coq [29], do sometimes solve constraints even if there is not a most general solution. This approach can still be predictable, if programmers are familiar with the workings of the unification algorithm. Ziliani and Sozeau [29] argue that even if there is no most general solution there could still be a most *natural* one, but they also claim that the algorithm used in one version of Coq is unpredictable, and suggest changes to it.

In another approach solutions are only chosen if they are unique or most general. This approach is for instance taken by Agda (ignoring bugs). This has the advantage that one does not need to know details of the unification algorithm in order to understand a piece of code. A drawback is that sometimes "natural" solutions are not found, but on the other hand there is perhaps less need to make the unification algorithm predictable to users.

In §2.4 we discuss an example, based on "real" code, which is handled by Agda, but for which Coq, Idris, Lean and Matita all fail. Agda handles this example because it can start unifying terms before their types are known to be

equal, and in the case of this example work on the terms uncovers information which is used to unify the types. Unfortunately this part of Agda is buggy [6, 25, 27].

Gundry and McBride's technique with twin types and twin variables [10] also makes it possible to start unifying terms before their types are known to be equal, and furthermore Gundry presents correctness proofs [9]. However, so far this technique has not been tested in practice as part of a full type checker.

We believe that these are our main contributions:

- We present the first implementation of a twin type approach in a type checker for a dependently typed language (we adapted a pre-existing type checker for a tiny variant of Agda, called Tog [19]).
- We use a reformulation of the approach of Gundry and McBride without twin variables. The language that is presented in §3.1 below is intended to be closer to the language used internally in Agda, and thus the technique might be a little easier to adopt, avoiding modifying the grammar of terms or the algorithms that manipulate them.
- The approach can handle some code that is handled by Agda (2.6.0.1) but not by Coq (8.11.0), Idris (1.3.2), Lean (3.4.2) or Matita (0.99.3), see §2.4 and §3.5.
- A small case study (§4) based on "real" code that a previous version of Agda struggled with suggests that the approach might be feasible in practice: the performance is similar to that of Agda 2.6.0.1.
- We introduce a notion of heterogeneous equality (3.2) that enables a type-agnostic syntactic equality rule. Benchmarks suggest that use of this rule can, at least in some cases, improve performance (4.2).

The text is based on the first author's forthcoming licentiate thesis [14]. The thesis contains detailed proofs of the main results stated in this paper. However, note that the type theory uses a type-in-type axiom. The results have been proved under the assumption that certain properties hold, and these properties may not hold for the type theory as given, with the type-in-type axiom. Furthermore the proofs are not machine-checked, and due to their size we acknowledge that it is likely that they contain errors. For these reasons we do not claim that the presented approach is correct. We have also not proved that the program used for the benchmarks implements the theory correctly.

2 Unification

Let us begin by discussing our approach to unification in more detail. For the examples in this section we use a syntax similar to that of Agda.

2.1 Unique Solutions

As an example, consider the function replicate, which returns a vector (a list of fixed length) repeating an element:

```
replicate : {n : Nat} → Int → Vec n Int
```

For instance, replicate (-4) : Vec 3 Int gives $[-4, -4, -4]$, and replicate $\{n = 5\}$ (-4) gives $[-4, -4, -4, -4, -4]$.

The first (implicit) argument to replicate determines the length of the resulting vector. If this argument is not given explicitly, then an attempt is made to infer it from the context. Such a context could be provided by the function rotate90 or the command print:

```
rotate90 : Vec 2 Int → Vec 2 Int
print    : {n : Nat} → Vec n Int → IO ()
```

The function rotate90 rotates a vector 90° clockwise (e.g. rotate90 $[1, 2]$ gives $[-2, 1]$). The function print outputs a vector to the terminal.

Now consider the following program:

```
main : IO ()
main = do
  print (rotate90 (replicate 1))
  print (replicate 6)
```

It is easy to figure out that the vector returned by replicate 1 should have length 2, and thus print (rotate90 (replicate 1)) outputs $[-1, 1]$. However, there is nothing that constrains the length of the vector returned by replicate 6. For underspecified programs of this kind we expect that the type checker does not fill in arbitrary type-correct terms: we do not want the type checker to make unforced choices on behalf of the programmer. (We do not claim that any proof assistant in use today would do so in this particular case, this is a contrived example used to illustrate our point.)

Avoiding ambiguity is not only important for the well-defined behaviour of programs, but can also be important in statements of theorems and—in proof-relevant settings—in the bodies of proofs.

2.2 Constraints and Solutions

In our development, all judgments about terms and types are formulated inside a signature. A signature Σ represents atom declarations ($\mathbb{a} : A$, corresponding to an Agda postulate or a Coq Axiom), together with metavariable declarations (typically added by the type-checker in place of implicit arguments, or subterms of potentially mismatched types). Each metavariable is either declared ($\alpha : A$), or declared and instantiated ($\alpha := t : A$):

$$\Sigma ::= \cdot \mid \Sigma, \mathbb{a} : A \mid \Sigma, \alpha : A \mid \Sigma, \alpha := t : A$$

According to Mazzoli and Abel [18] dependent type checking with metavariables can be reduced to solving a set of higher-order unification constraints of the form $\Gamma_i \vdash t_i : A_i \equiv^? u_i : B_i$ with a common signature Σ. All of the resulting constraints are well-typed in Σ: that is, $\Sigma; \Gamma_i \vdash t_i : A_i$ and $\Sigma; \Gamma_i \vdash u_i : B_i$.

Solving a higher unification problem is an assignment of terms of the appropriate type to all of the uninstantiated metavariables in Σ yielding a signature Σ', in such a way

that all the constraints hold (that is, for each constraint $\Gamma_i \vdash t_i : A_i \equiv^? u_i : B_i$, we have $\Sigma'; \Gamma_i \vdash A_i \equiv B_i$ **type** and $\Sigma'; \Gamma_i \vdash t_i \equiv u_i : A_i$).

When finding solutions, we have two conflicting goals: we want to ensure that the algorithm can tackle as many unification problems as possible, but at the same time, we want to avoid producing solutions that are ill-typed.

2.3 Goal #1: Well-Typedness

As described in the introduction, when implementing an algorithm to solve unification problems, we want both the resulting solutions and intermediate results to always be well-typed. However, pervasively checking that the terms involved in any operation are well-typed would incur an additional cost for the type-checker. Instead, we assume that well-formed constraints are derived from the type-checking problem, and aim to maintain the well-typedness of terms as an invariant.

The current implementation of Agda will violate this well-typedness invariant in some cases, leading to ill-typed solutions. Some ill-typed programs cause the type checker to crash [25], and workarounds were even added to prevent crashes for some well-typed programs [6]. Consider the following problem derived from a currently open bug report [25]:

Example 2.1. For simplicity we assume that we have a data type Nat of natural numbers, and functions F and f defined in the following way (using Agda-like notation):

$$\begin{array}{ll} F : \text{Bool} \to \text{Set} & f : (x : \text{Bool}) \to F\,x \to \text{Nat} \\ F\ \ \text{false} = \text{Bool} & f\ \ \text{false}\ \ \text{false} = 0 \\ F\ \ \text{true}\ \ = \text{Nat} & f\ \ \text{false}\ \ \text{true}\ \ = 1 \\ & f\ \ \text{true}\ \ \ x\ \ \ \ = 2 \end{array}$$

Given the signature $\Sigma \stackrel{\text{def}}{=} \mathbb{D} : \text{Nat} \to \text{Set}, \alpha : \text{Nat} \to \text{Set}, \beta : \text{Nat} \to \text{Bool}$ we can form the following well-formed constraints:

$$\begin{array}{l} \Sigma; \cdot \vdash (x : \text{Nat}) \to \alpha\,x : \text{Set} \equiv^? \\ \qquad (x : F\,(\beta\,0)) \to \mathbb{D}\,(f\,(\beta\,0)\,x) : \text{Set} \\ \cdot \vdash \beta : \text{Nat} \to \text{Bool} \equiv^? \lambda x.\text{false} : \text{Nat} \to \text{Bool} \qquad \blacksquare \end{array}$$

When tackling the constraints in Example 2.1, Agda will first make $\alpha\,x$ and $\mathbb{D}\,(f\,(\beta\,0)\,x)$ equal by instantiating α (of type Nat $\to$ Set), to $\lambda x.\mathbb{D}\,(f\,(\beta\,0)\,x)$ (of type $F\,(\beta\,0) \to$ Set). Then it will solve the second constraint by instantiating β to $\lambda x.$false, rendering the instantiation of α ill-typed. The end result is the ill-typed solution $\Sigma' \stackrel{\text{def}}{=} \mathbb{D} : \text{Nat} \to \text{Set}, \alpha := \lambda x.\mathbb{D}\,(f\,\text{false}\,x) : \text{Nat} \to \text{Set}, \beta := \lambda x.\text{false} : \text{Bool} \to \text{Set}$.

One may attempt to preserve well-typedness invariants by ensuring that terms have equal types before unifying them, as is done by Tog [19]. However, this may prove too restrictive in practice, as we explain in the following section.

2.4 Goal #2: Out of Order Unification

McBride [20] shows how to embed some constructs of type-theory into type-theory itself. A program based on this work is used in §4.2. This program yields unification constraints in which metavariables appear both in the term and the type. The following example shows a reduced version of one of these constraints.

Example 2.2. We use Agda syntax to define an inductive data type BoolOp of optional booleans and a function get:

$$\begin{array}{ll} \text{data BoolOp : Set where} & \text{get : BoolOp} \to \text{Bool} \\ \quad \text{None : BoolOp} & \text{get None = true} \\ \quad \text{Some : Bool} \to \text{BoolOp} & \text{get (Some } x\text{) = } x \end{array}$$

We then define the signature $\Sigma \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \alpha : \text{Bool} \to \text{BoolOp}$, and the following well-formed constraint:

$$\begin{array}{l} \Sigma; x : \text{Bool} \vdash \lambda y.\text{None} : \mathbb{F}\,(\text{get}\,(\alpha\,x)) \to \text{BoolOp} \equiv^? \\ \qquad \lambda y.(\alpha\,x) : \mathbb{F}\,\text{true} \to \text{BoolOp} \qquad \blacksquare \end{array}$$

Solving the problem in Example 2.2 entails finding a term t such that $\Sigma' \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \alpha := t : \text{Bool} \to \text{BoolOp}$ is well-formed and the following two equalities hold:

$$\begin{array}{l} \Sigma'; x : \text{Bool} \vdash \mathbb{F}\,(\text{get}\,(\alpha\,x)) \to \text{BoolOp} \equiv \\ \qquad \mathbb{F}\,\text{true} \to \text{BoolOp} : \text{Set} \end{array} \tag{1}$$

$$\begin{array}{l} \Sigma'; x : \text{Bool} \vdash (\lambda y.\text{None}) \equiv \\ \qquad (\lambda y.(\alpha\,x)) : \mathbb{F}\,(\text{get}\,(\alpha\,x)) \to \text{BoolOp} \end{array} \tag{2}$$

The first equality (1) does not contain enough information to determine t; both $t = \lambda x.$None and $t = \lambda x.$Some true are possible.

The second equality (2) does contain enough information (the only solution is $t = \lambda x.$None), but one of the terms (namely, $(\lambda y.$None$)$) is being compared at a type ($\mathbb{F}$ (get $(\alpha\,x)) \to$ BoolOp) which is not its original one ($\mathbb{F}$ true $\to$ BoolOp). In general, tackling constraints with potentially incorrect types leads to the issues described in §2.3.

In our tests, Agda (2.6.0.1) could instantiate α, but none of Coq (8.11.0), Idris (1.3.2), Lean (3.4.2), or Matita (0.99.3) could. The difficulty lies in formulating the second equality (2) as a constraint in which both sides are well-typed, and being able to manipulate this constraint in order to obtain a solution for α. This is where twin types come in.

2.5 Twin Types

Gundry and McBride [9, 10] propose the use of twin types, which are two types separated by a dagger (e.g. $x : A_1 \ddagger A_2$). If a variable has a twin type, then the type for a specific use of the variable is indicated by an annotation (like in $\acute{x}$ or $\grave{x}$). We propose a variant of this approach where the type of each occurrence of a variable is determined by the side of the constraint it occurs on, thus dispensing with the annotations. This change might make it a little easier to use this approach in existing type checkers, by hopefully restricting

the need to deal with twin variables to the unification algorithm.

In our setting, each original well-formed constraint of the form $\Gamma \vdash t : A \equiv^? u : B$ is elaborated into two well-formed internal constraints, one for the types ($\Gamma \ddagger \Gamma \vdash A \cong^? B :$ Set‡Set) and one for the terms ($\Gamma \ddagger \Gamma \vdash t \cong^? u : A \ddagger B$). The resulting constraints form an internal unification problem.

Definition 2.3 (Internal unification problem)**.** An internal (unification) problem is of the form $\Sigma; \vec{\mathcal{C}}$, where for each $\mathcal{C} \in \vec{\mathcal{C}}$, $\mathcal{C}$ has the form $\Gamma_1 \ddagger \Gamma_2 \vdash t \cong^? u : A \ddagger B$. The problem is well-formed if the signature is well-formed, and, for each constraint $\Gamma_1 \ddagger \Gamma_2 \vdash t \cong^? u : A \ddagger B \in \vec{\mathcal{C}}$, Γ_1 and Γ_2 have the same length, and we have $\Sigma; \Gamma_1 \vdash t : A$ and $\Sigma; \Gamma_2 \vdash u : B$.

Notation (Twin context)**.** Given two contexts $\Gamma_1 = A_1, \ldots, A_n$ and $\Gamma_2 = B_1, \ldots, B_n$, we may write $\Gamma_1 \ddagger \Gamma_2$ as $A_1 \ddagger B_1, \ldots, A_n \ddagger B_n$. Furthermore, for clarity, we may name the variables as $x_1 : A_1 \ddagger B_1, \ldots, x_n : A_n \ddagger B_n$.

For instance, the unification problem in Example 2.2 corresponds to the well-formed internal problem $\Sigma; \mathcal{C}_1, \mathcal{C}_2$, with $\mathcal{C}_1$ and $\mathcal{C}_2$ defined as follows:

$$\mathcal{C}_1 \stackrel{\text{def}}{=} x : \text{Bool}\ddagger\text{Bool} \vdash \mathbb{F}\ (\text{get}\ (\alpha\, x)) \to \text{BoolOp} \cong^? \mathbb{F}\ \text{true} \to \text{BoolOp} : \text{Set}\ddagger\text{Set}$$

$$\mathcal{C}_2 \stackrel{\text{def}}{=} x : \text{Bool}\ddagger\text{Bool} \vdash \lambda y.\text{None} \cong^? \lambda y.(\alpha\, x) : (\mathbb{F}\ (\text{get}\ (\alpha\, x)) \to \text{BoolOp})\ddagger(\mathbb{F}\ \text{true} \to \text{BoolOp})$$

Because all constraints are well-formed (in particular, $\mathcal{C}_2$), this constraint can be tackled immediately. By instantiating $\alpha := \lambda y.\text{None} : \text{Bool} \to \text{BoolOp}$, both sides of the constraint become *heterogeneously* equal, as defined in §3.2. The exact sequence of rules applied in this case is given in §3.5.

3 Dependent Type Checking Using Twin Types

In this section we describe the details of our approach from a more technical point of view.

3.1 Language

We describe a type theory that is based on our implementation. It is an intensional type theory with Π-types, Σ-types, η-equality, metavariables and large elimination from Bool. The goal is to have a theory with a small number of constructs which still gives rise to some of the type-checking problems that occur in a full-fledged proof assistant.

For simplicity we use a single universe Set with a type-in-type axiom. Below we state some properties without proofs (e.g. Statement 1). Some of these properties may not hold in the presence of the type-in-type axiom, but we hope they would hold in a properly stratified version of the theory.

The main judgment of the type theory is the typing judgment, which is of the form $\Sigma; \Gamma \vdash t : A$: a term t has type A in context Γ and signature Σ. We represent variables using de Bruijn indices $(0, 1, 2, \ldots)$ in order to stay close to our implementation. Indices are manipulated using weakenings ($t^{(+n)}$, increment all indices by n) and renamings ($t[\vec{x} \mapsto \vec{y}]$, replace each variable in x with the corresponding variable in y, where x and y are lists of indices of the same length). Terms are in β-normal form in order to simplify type-checking. The substitution ($t[u/x]$) and application ($t \ @\ \vec{e}$) operations are hereditary, that is, they directly compute terms without β-redexes. For instance, $(x\, y_1\, y_2)[\lambda z_1.\lambda z_2.z_1/x] \stackrel{\text{def}}{=} (\lambda z_1.\lambda z_2.z_1)\ @\ y_1\, y_2 \stackrel{\text{def}}{=} (\lambda z_2.z_1)[y_1/z_1]\ @\ y_2 \stackrel{\text{def}}{=} (\lambda z_2.y_1)\ @\ y_2 \stackrel{\text{def}}{=} y_1[y_2/z_2] \stackrel{\text{def}}{=} y_2$. Hereditary substitution and application are deterministic, but are not guaranteed to terminate due to type-in-type. When we write $t[u/x]$ or $t\ @\ \vec{e}$ we implicitly assume that the process terminates for that particular choice of terms. The notation $t[u]$ is shorthand for $t[u/0]$.

Definition 3.1 (Signature)**.** A well-formed signature Σ is defined as follows:

$$\frac{}{\cdot\ \textbf{sig}}\ \text{EMPTY}$$

$$\frac{\Sigma\ \textbf{sig} \qquad \mathbb{a}\text{ is fresh in }\Sigma \qquad \Sigma; \cdot \vdash A\ \textbf{type}}{\Sigma, \mathbb{a} : A\ \textbf{sig}}\ \text{ATOM-DECL}$$

$$\frac{\Sigma\ \textbf{sig} \qquad \alpha\text{ is fresh in }\Sigma \qquad \Sigma; \cdot \vdash A\ \textbf{type}}{\Sigma, \alpha : A\ \textbf{sig}}\ \text{META-DECL}$$

$$\frac{\Sigma, \alpha : A\ \textbf{sig} \qquad \Sigma; \cdot \vdash t : A}{\Sigma, \alpha := t : A\ \textbf{sig}}\ \text{META-INST}$$

Definition 3.2 (Context)**.** A context Γ is defined as follows:

$$\frac{\Sigma\ \textbf{sig}}{\Sigma \vdash \cdot\ \textbf{ctx}} \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx} \qquad \Sigma; \Gamma \vdash A\ \textbf{type}}{\Sigma \vdash \Gamma, A\ \textbf{ctx}}$$

Definition 3.3 (Terms and neutral terms)**.** Terms ($t, u, \ldots, A, B, \ldots$ or f) and the typing judgment $\Sigma; \Gamma \vdash t : A$ are defined in Figure 1. Those terms of the form $h\,\vec{e}$, where each $e \in \vec{e}$ is of the form $e = t \mid .\pi_1 \mid .\pi_2$, are called neutral terms, and denoted by f.

Notation. For the sake of clarity, we may sometimes use $(x : A) \to B$ or $A \to B$ to denote $\Pi A B$; and $(x : A) \times B$ or $A \times B$ to denote $\Sigma A B$.

Lemma 3.4 (Piecewise well-formedness of the typing judgment)**.** *If* $\Sigma; \Gamma \vdash t : A$, *then* $\Sigma \vdash \Gamma$ **ctx** *and* Σ **sig**.

Proof. By induction on the derivation, noting that if $\Sigma \vdash \Gamma, \Gamma'$ **ctx** then $\Sigma \vdash \Gamma$ **ctx** and Σ **sig**. □

Definition 3.5 (Judgmental equality of terms, types and contexts)**.** $\Sigma; \Gamma \vdash t \equiv u : A$ means that the terms t and u are judgmentally equal at type A. Some, but not all, rules of this relation are given in Figure 2.

We state that the relations in Definition 3.5 are equivalence relations:

$$\frac{\Sigma \vdash \Gamma\ \textbf{ctx}}{\Sigma;\Gamma \vdash \mathsf{Bool} : \mathsf{Set}}\ \textsc{bool} \qquad \frac{\Sigma;\Gamma \vdash A : \mathsf{Set} \qquad \Sigma;\Gamma, A \vdash B : \mathsf{Set}}{\Sigma;\Gamma \vdash \Pi A B : \mathsf{Set}}\ \textsc{pi}$$

$$\frac{\Sigma;\Gamma \vdash A : \mathsf{Set} \qquad \Sigma;\Gamma, A \vdash B : \mathsf{Set}}{\Sigma;\Gamma \vdash \Sigma A B : \mathsf{Set}}\ \textsc{sigma} \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx}}{\Sigma;\Gamma \vdash \mathsf{Set} : \mathsf{Set}}\ \textsc{set} \qquad \frac{\Sigma;\Gamma \vdash A : \mathsf{Set}}{\Sigma;\Gamma \vdash A\ \textbf{type}}\ \textsc{type}$$

$$\frac{\Sigma \vdash \Gamma\ \textbf{ctx}}{\Sigma;\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \textsc{true} \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx}}{\Sigma;\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \textsc{false} \qquad \frac{\Sigma;\Gamma, A \vdash t : B}{\Sigma;\Gamma \vdash \lambda.t : \Pi A B}\ \textsc{abs}$$

$$\frac{\Sigma;\Gamma \vdash t : A \qquad \Sigma;\Gamma, A \vdash B\ \textbf{type} \qquad \Sigma;\Gamma \vdash u : B[t]}{\Sigma;\Gamma \vdash \langle t, u\rangle : \Sigma A B}\ \textsc{pair} \qquad \frac{\Sigma;\Gamma \vdash t : A \qquad \Sigma;\Gamma \vdash A \equiv B\ \textbf{type}}{\Sigma;\Gamma \vdash t : B}\ \textsc{conv}$$

$$\frac{\Sigma \vdash \Gamma\ \textbf{ctx} \qquad \Gamma = \Gamma_1, A, \Gamma_2 \qquad n = |\Gamma_2|}{\Sigma;\Gamma \vdash n \Rightarrow A^{(+(n+1))}}\ \textsc{var} \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx} \qquad \mathbb{a} : A \in \Sigma}{\Sigma;\Gamma \vdash \mathbb{a} \Rightarrow A}\ \textsc{atom}$$

$$\frac{\Sigma \vdash \Gamma\ \textbf{ctx} \qquad \alpha : A \in \Sigma}{\Sigma;\Gamma \vdash \alpha \Rightarrow A}\ \textsc{meta}_1 \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx} \qquad \alpha := t : A \in \Sigma}{\Sigma;\Gamma \vdash \alpha \Rightarrow A}\ \textsc{meta}_2 \qquad \frac{\Sigma \vdash \Gamma\ \textbf{ctx}}{\Sigma;\Gamma \vdash \mathsf{if} \Rightarrow T_{\mathsf{if}}}\ \textsc{if}$$

$$\frac{\Sigma;\Gamma \vdash h \Rightarrow A}{\Sigma;\Gamma \vdash h : A}\ \textsc{head} \qquad \frac{\Sigma;\Gamma \vdash f : \Sigma A B}{\Sigma;\Gamma \vdash f\,.\pi_1 : A}\ \textsc{proj1} \qquad \frac{\Sigma;\Gamma \vdash f : \Sigma A B}{\Sigma;\Gamma \vdash f\,.\pi_2 : B[f\,.\pi_1]}\ \textsc{proj2}$$

$$\frac{\Sigma;\Gamma \vdash f : \Pi A B \qquad \Sigma;\Gamma \vdash t : A}{\Sigma;\Gamma \vdash f\,t : B[t]}\ \textsc{app}$$

Figure 1. Typing rules for terms. $T_{\mathsf{if}} \stackrel{\text{def}}{=} \Pi(\Pi\mathsf{Bool}\mathsf{Set})(\Pi\mathsf{Bool}(\Pi(1\,\mathsf{true})(\Pi(2\,\mathsf{false})(3\,2))))$, or, informally, $(X : \mathsf{Bool} \to \mathsf{Set}) \to (z : \mathsf{Bool}) \to (X\,\mathsf{true}) \to (X\,\mathsf{false}) \to (X\,z)$.

Statement 1. *The relations* $\Sigma;\Gamma \vdash _ \equiv _\ \textbf{type}$, $\Sigma \vdash _ \equiv _\ \textbf{ctx}$ *and the full relation* $\Sigma;\Gamma \vdash _ \equiv _ : A$ *are reflexive, symmetric, and transitive over the sets* $\{A \mid \Sigma;\Gamma \vdash A\ \textbf{type}\}$, $\{\Gamma \mid \Sigma \vdash \Gamma\ \textbf{ctx}\}$ *and* $\{t \mid \Sigma;\Gamma \vdash t : A\ \textbf{type}\}$, *respectively.*

Notation. We use $\textsc{fv}(t)$ to denote the set of free variables of t (e.g. $\textsc{fv}(\lambda.(3\,0)) = \{2\}$). Also, $\textsc{fv}(\Delta \vdash t : U)$ denotes the set of free variables in Δ, t and U, excluding those introduced by Δ (e.g. $\textsc{fv}(\mathbb{A}, \mathbb{B}\,1 \vdash 9\,0 : \mathbb{C}) = \{7, 0\}$).

Notation. We use $\textsc{metas}(t)$ to denote the set of metavariables occurring in a term (e.g. $\textsc{metas}(\lambda.\lambda.(\alpha\,\mathbb{b})) = \{\alpha\}$), and $\textsc{consts}(t)$ to denote the set of atoms and metavariables occurring in a term (e.g. $\textsc{consts}(\lambda.\lambda.(\alpha\,\mathbb{b})) = \{\alpha, \mathbb{b}\}$). Also, given a signature Σ, we define $\textsc{decls}(\Sigma) = \{\mathbb{a} \mid \mathbb{a} : A \in \Sigma\} \cup \{\alpha \mid \alpha : A \in \Sigma \vee \alpha := t : A \in \Sigma\}$.

3.2 A Heterogeneous Notion of Equality

As explained in §2.4 we want to unify terms before their types are known to be equal. To make sense of what it means for two terms of potentially different types to be equal, we introduce a notion of *heterogeneous equality.*

Definition 3.6 (Heterogeneous equality)**.** Let t and u be terms such that $\Sigma;\Gamma_1 \vdash t : A$ and $\Sigma;\Gamma_2 \vdash u : B$. If there exists a term v such that $\Sigma;\Gamma_1 \vdash t \equiv v : A$, $\Sigma;\Gamma_2 \vdash u \equiv v : B$ and $\textsc{fv}(v) \subseteq \textsc{fv}(t) \cap \textsc{fv}(u)$ (the latter called *the interpolant property*), then we say that t and u are *heterogeneously equal* with witness v, and write $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash t \cong\!\{v\}\!\cong u : A \ddagger B$. We may also write $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash t \cong u : A \ddagger B$ if there is no need to refer to v.

Remark 1. The heterogeneous equality is reflexive (given $\Sigma;\Gamma \vdash t : A$, we have $\Sigma;\Gamma \ddagger \Gamma \vdash t \cong\!\{t\}\!\cong t : A \ddagger A$) and symmetric by definition.

Remark 2. If $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash t \cong\!\{v\}\!\cong u : A \ddagger B$, then by Definition 3.5 we have $\Sigma;\Gamma_1 \vdash v : A$ and $\Sigma;\Gamma_2 \vdash v : B$. However, this does not mean that $\Sigma \vdash \Gamma_1 \equiv \Gamma_2\,\textbf{ctx}$, or $\Sigma, \Gamma_1 \vdash A \equiv B\ \textbf{type}$. For example (using variable names for clarity), let $\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathsf{Set}, \mathbb{B} : \mathsf{Set}$, $A \stackrel{\text{def}}{=} \mathbb{A} \to \mathbb{A}$, $B \stackrel{\text{def}}{=} \mathbb{B}$, $\Gamma_1 \stackrel{\text{def}}{=} x : B, z : A$, $\Gamma_2 \stackrel{\text{def}}{=} x : A, z : B$. Then we have $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash \langle x, \lambda y.z\,y\rangle \cong\!\{\langle x, z\rangle\}\!\cong \langle \lambda y.x\,y, z\rangle : (B \times A) \ddagger (A \times B)$.

We use Definition 3.6 to formulate the soundness of the rules given in the following section (Definition 3.13), and, in particular, to describe the conditions under which metavariables are instantiated (Rule Schema 11).

3.3 A Rule Schema Toolkit

Each step of our unification algorithm consists of the application of a (rewrite) rule to a signature and/or one or more constraints, producing a new signature and new constraints.

Definition 3.7 (Rule)**.** A rule is a four-tuple of the form $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$, where Σ and Σ' are signatures, and $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ are lists of internal constraints. A rule schema is a family of rules.

In this section we list some of the rules that are used in the algorithm (we do not have room to list all of them).

$$\frac{\Sigma;\Gamma \vdash f : \Pi A B}{\Sigma;\Gamma \vdash f \equiv \lambda. f^{(+1)}\, 0 : \Pi A B}\ \textsc{eta-abs} \qquad \frac{\Sigma;\Gamma \vdash f : \Sigma A B}{\Sigma;\Gamma \vdash f \equiv \langle f\,.\pi_1, f\,.\pi_2\rangle : \Sigma A B}\ \textsc{eta-pair}$$

$$\frac{\Sigma;\Gamma \vdash \alpha\,\vec{e} : T \qquad \alpha \coloneqq t : A \in \Sigma \qquad \Sigma;\Gamma \vdash (t\ @\ \vec{e}) : T}{\Sigma;\Gamma \vdash \alpha\,\vec{e} \equiv (t\ @\ \vec{e}) : T}\ \textsc{delta-meta}$$

$$\frac{\Sigma;\Gamma \vdash \text{if}\ A\ \text{true}\ u_{\text{t}}\ u_{\text{f}}\,\vec{e} : T \qquad \Sigma;\Gamma \vdash (u_{\text{t}}\ @\ \vec{e}) : T}{\Sigma;\Gamma \vdash \text{if}\ A\ \text{true}\ u_{\text{t}}\ u_{\text{f}}\,\vec{e} \equiv (u_{\text{t}}\ @\ \vec{e}) : T}\ \textsc{delta-if-true}$$

$$\frac{\Sigma;\Gamma \vdash \text{if}\ A\ \text{false}\ u_{\text{t}}\ u_{\text{f}}\,\vec{e} : T \qquad \Sigma;\Gamma \vdash (u_{\text{f}}\ @\ \vec{e}) : T}{\Sigma;\Gamma \vdash \text{if}\ A\ \text{false}\ u_{\text{t}}\ u_{\text{f}}\,\vec{e} \equiv (u_{\text{f}}\ @\ \vec{e}) : T}\ \textsc{delta-if-false}$$

$$\frac{\Sigma;\Gamma \vdash t \equiv u : A \qquad \Sigma;\Gamma \vdash A \equiv B\ \textbf{type}}{\Sigma;\Gamma \vdash t \equiv u : B}\ \textsc{conv-eq}$$

$$\frac{\Sigma;\Gamma \vdash A \equiv B : \text{Set}}{\Sigma;\Gamma \vdash A \equiv B\ \textbf{type}}\ \textsc{type-eq} \qquad \frac{}{\Sigma \vdash \cdot \equiv \cdot}\ \textsc{ctx-empty-eq} \qquad \frac{\Sigma \vdash \Gamma_1 \equiv \Gamma_2\ \textbf{ctx} \qquad \Sigma;\Gamma_1 \vdash A_1 \equiv A_2\ \textbf{type}}{\Sigma \vdash \Gamma_1, A_1 \equiv \Gamma_2, A_2\ \textbf{ctx}}\ \textsc{ctx-var-eq}$$

Figure 2. Judgmental equality for terms, types and contexts

Notation. When writing down a list of constraints, we may use $\wedge$ as a separator (instead of a comma), and $\square$ to stand for an empty such list.

Rule Schema 1 (Syntactic equality). $\Sigma;\Gamma\ddagger\Gamma' \vdash t \approx t : A\ddagger A' \rightsquigarrow \Sigma;\square$

Rule Schema 2 (λ-abstraction). $\Sigma;\Gamma\ddagger\Gamma' \vdash \lambda.t \approx \lambda.u : \Pi A B\ddagger\Pi A' B' \rightsquigarrow \Sigma;\Gamma\ddagger\Gamma', A\ddagger A' \vdash t \approx u : B\ddagger B'$

Rule Schema 3 (Term conversion). $\Sigma;\Gamma\ddagger\Gamma' \vdash t \approx u : A\ddagger A' \rightsquigarrow \Sigma;\Gamma\ddagger\Gamma' \vdash t' \approx u : A\ddagger A'$ **where** $\Sigma;\Gamma \vdash t \equiv t' : A$ **and** $\textsc{fv}(t) \supseteq \textsc{fv}(t')$.

Rule Schema 4 (Injectivity of Π). $\Sigma;\Gamma\ddagger\Gamma' \vdash \Pi A B \approx \Pi A' B' : \text{Set}\ddagger\text{Set} \rightsquigarrow \Sigma;\Gamma\ddagger\Gamma' \vdash A \approx A' : \text{Set}\ddagger\text{Set} \wedge \Gamma\ddagger\Gamma', A\ddagger A' \vdash B \approx B' : \text{Set}\ddagger\text{Set}$

Rule Schema 5 (Injectivity of Σ). *Analogous to Rule Schema 4, replacing* Π *by* Σ.

Rule Schema 6 (Constraint symmetry). $\Sigma;\Gamma\ddagger\Gamma' \vdash t \approx u : A\ddagger A' \rightsquigarrow \Sigma;\Gamma'\ddagger\Gamma \vdash u \approx t : A'\ddagger A$

Rule Schema 7 (Type and context conversion). $\Sigma;\Gamma\ddagger\Gamma' \vdash t \approx u : A\ddagger A' \rightsquigarrow \Sigma;\Gamma_0\ddagger\Gamma' \vdash t \approx u : A_0\ddagger A'$ **where** $\Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0\ \textbf{ctx}$

Rule Schema 8 (Pairs). $\Sigma;\Gamma\ddagger\Gamma' \vdash \langle t_1, t_2\rangle \approx \langle u_1, u_2\rangle : \Sigma A B\ddagger\Sigma A' B' \rightsquigarrow \Sigma;\Gamma\ddagger\Gamma' \vdash t_1 \approx u_1 : A\ddagger A' \wedge \Gamma\ddagger\Gamma' \vdash t_2 \approx u_2 : B[t_1]\ddagger B'[u_1]$

Definition 3.8 (Strongly neutral term). A neutral term f is strongly neutral if f is of the form $x\,\vec{e}$, $\mathbb{c}\,\vec{e}$, or if $\vec{e}$, where in the last case either the length of $\vec{e}$ is less than 2, or the second element in $\vec{e}$ (the branching condition) is a strongly neutral term.

Rule Schema 9 (Rigid-rigid unification). *If* $h\,\vec{e}$ *and* $h\,\vec{e'}$ *are strongly neutral;* $|\vec{e}| = |\vec{e'}| = n$*;* $J \subseteq \{1, ..., n\}$*; for each* $i \in \{1, ..., n\} - J$ *either* $e_i = e'_i = .\pi_1$ *or* $e_i = e'_i = .\pi_2$*; and for each* $i \in J$ *there exist* $t_i, u_i, B_i, C_i, B'_i, C'_i$ *such that* $e_i = t_i$*,* $e'_i = u_i$*,* $\Sigma;\Gamma \vdash h\,\vec{e}_{1,...,i-1} : \Pi B_i C_i$ *and* $\Sigma;\Gamma' \vdash h\,\vec{e'}_{1,...,i-1} : \Pi B'_i C'_i$*, then:*

$$\Sigma;\Gamma\ddagger\Gamma' \vdash h\,\vec{e} \approx h\,\vec{e'} : T\ddagger T' \rightsquigarrow \Sigma; \bigwedge_{i\in J} \Gamma\ddagger\Gamma' \vdash t_i \approx u_i : B_i\ddagger B'_i$$

The following unification rules modify the signature by i) adding new metavariable declarations, ii) instantiating existing metavariables and iii) reordering and normalizing terms in the signature. We call the result of these operations a signature extension.

Definition 3.9 (Signature extension: $\Sigma \sqsubseteq \Sigma'$). We say that Σ' extends Σ (written $\Sigma' \sqsupseteq \Sigma$ or $\Sigma \sqsubseteq \Sigma'$), iff Σ **sig**, Σ' **sig** and $\Sigma \sqsubseteq' \Sigma'$, where $\sqsubseteq'$ is defined inductively:

i) $\Sigma_1, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha : A, \Sigma_2$
ii) $\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha \coloneqq t : A, \Sigma_2$
iii) $\Sigma_1 \sqsubseteq' \Sigma_3$ **if** $\Sigma_1 \sqsubseteq \Sigma_2$ **and** $\Sigma_2 \sqsubseteq \Sigma_3$
iv) $\Sigma_1, \mathbb{c} : A, \Sigma_2 \sqsubseteq' \Sigma_1, \mathbb{c} : A', \Sigma_2$ **if** $\Sigma_1; \cdot \vdash A \equiv A'$ **type**
v) $\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha : A', \Sigma_2$ **if** $\Sigma_1; \cdot \vdash A \equiv A'$ **type**
vi) $\Sigma_1, \alpha \coloneqq t : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha \coloneqq t : A', \Sigma_2$ **if** $\Sigma_1; \cdot \vdash A \equiv A'$ **type**
vii) $\Sigma_1, \alpha \coloneqq t : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha \coloneqq t' : A, \Sigma_2$ **if** $\Sigma_1; \cdot \vdash t \equiv t' : A$
viii) $\Sigma \sqsubseteq' \Sigma'$ **if** Σ' is a (possibly trivial) reordering of Σ

In particular, we allow reordering and normalizing elements in the signature:

Rule Schema 10 (Signature conversion). $\Sigma;\square \rightsquigarrow \Sigma';\square$ **where** $\Sigma \sqsubseteq \Sigma'$ **and** $\Sigma \sqsupseteq \Sigma'$

Signature extensions have been defined with the aim of justifying the following statement:

Statement 2 (Signature extension)**.** *If* $\Sigma \vdash \Gamma$ **ctx**, $\Sigma;\Gamma \vdash A$ **type**, $\Sigma;\Gamma \vdash t : A$, $\Sigma;\Gamma \vdash A \equiv B$ **type** *or* $\Sigma;\Gamma \vdash t \equiv u : A$ *and* $\Sigma' \sqsupseteq \Sigma$, *then* $\Sigma' \vdash \Gamma$ **ctx**, $\Sigma';\Gamma \vdash A$ **type**, $\Sigma';\Gamma \vdash t : A$, $\Sigma';\Gamma \vdash A \equiv B$ **type** *or* $\Sigma';\Gamma \vdash t \equiv u : A$ *respectively.*

Metavariable Instantiation

Before instantiating a metavariable we check that the types of the two sides are compatible. However, we do not want to do this for irrelevant entries in the context. For this reason we generalize the heterogeneous equality to contexts. (Gundry discusses a similar idea [9, p. 69], but does not formalise it.)

Definition 3.10 (Heterogeneous context equality for sets of variables)**.** Two well-formed contexts $\Sigma \vdash \Gamma_1$ **ctx** and $\Sigma \vdash \Gamma_2$ **ctx** are heterogeneously equal in the signature Σ for the sets of variables X_1 and X_2 (written $\Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{X_1,X_2} \Gamma_2$) iff:

$$\frac{\Sigma \ \mathbf{sig}}{\Sigma \vdash \cdot \cong_{\emptyset,\emptyset} \cdot}\ \text{EMPTY}$$

$$\frac{0 \notin X_1 \cup X_2 \qquad \Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{X_1-1,X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \cong\{\Gamma, \mathrm{Set}\}\cong_{X_1,X_2} \Gamma_2, A_2}\ \text{UNUSED}$$

$$\frac{0 \in X_2 - X_1 \qquad \Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{X_1-1,(X_2-1)\cup\mathrm{FV}(A_2)} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \cong\{\Gamma, A_2\}\cong_{X_1,X_2} \Gamma_2, A_2}\ \text{USED-R}$$

$$\frac{0 \in X_1 - X_2 \qquad \Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{(X_1-1)\cup\mathrm{FV}(A_1),X_2-1} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \cong\{\Gamma, A_1\}\cong_{X_1,X_2} \Gamma_2, A_2}\ \text{USED-L}$$

$$\frac{\begin{array}{c} 0 \in X_1 \cap X_2 \\ \Sigma;\Gamma_1\ddagger\Gamma_2 \vdash A_1 \cong\{A\}\cong A_2 : \mathrm{Set}\ddagger\mathrm{Set} \\ \Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{(X_1-1)\cup\mathrm{FV}(A_1),(X_2-1)\cup\mathrm{FV}(A_2)} \Gamma_2 \end{array}}{\Sigma \vdash \Gamma_1, A_1 \cong\{\Gamma, A\}\cong_{X_1,X_2} \Gamma_2, A_2}\ \text{USED}$$

(Given $X \subseteq \mathbb{N}$, let $(X-1) \stackrel{\text{def}}{=} \{x-1 | x \in X, x > 0\}$.)

Here are two more statements about the theory:

Statement 3 (Type of unused variables)**.** *If* $\Sigma;\Gamma \vdash B$ **type** *and* $\Sigma;\Gamma, A, \Delta \vdash t : T$ *with* $0 \notin \mathrm{FV}(\Delta \vdash t : T)$, *then* $\Sigma;\Gamma, B, \Delta \vdash t : T$. *This property generalizes to other judgments.*

Statement 4 (Context conversion)**.** *If* $\Sigma;\Gamma \vdash B$ **type** *and* $\Sigma;\Gamma, A, \Delta \vdash t : T$ *with* $\Sigma;\Gamma \vdash A \equiv B$ **type**, *then* $\Sigma;\Gamma, B, \Delta \vdash t : T$. *This property generalizes to other judgments.*

Lemma 3.11 (Typing in heterogeneously equal contexts)**.** *Let* t *and* u *be terms such that* $\Sigma;\Gamma_1 \vdash t : B_1$, $\Sigma;\Gamma_2 \vdash u : B_2$, *with* $|\Gamma_1| = |\Gamma_2|$.

Assume that we have (i) $\Sigma;\Gamma_1\ddagger\Gamma_2 \vdash B_1 \equiv\{B\}\equiv B_2 : \mathrm{Set}\ddagger\mathrm{Set}$ *and (ii)* $\Sigma \vdash \Gamma_1 \equiv\{\Gamma\}\equiv_{\mathrm{FV}(t)\cup\mathrm{FV}(B_1),\mathrm{FV}(u)\cup\mathrm{FV}(B_2)} \Gamma_2$.

Then $\Sigma;\Gamma \vdash t : B$ *and* $\Sigma;\Gamma \vdash u : B$.

Proof sketch. By induction on the derivation of (ii), using Statements 3 and 4 and the interpolant property from Definition 3.6. (See the first author's licentiate thesis [?] for a detailed proof.) ◻

Notation. We use "λ^n." to denote n copies of the binder "λ.", and "$\vec{x}^n$" to denote a vector of any n variables.

Rule Schema 11 (Metavariable instantiation)**.** $\Sigma;\Gamma_1\ddagger\Gamma_2 \vdash \alpha\,\vec{x}^n \approx t : B_1\ddagger B_2 \rightsquigarrow \Sigma;\square$ **where**
$\Sigma = \Sigma_1, \alpha : A, \Sigma_2$, *with* $\mathrm{CONSTS}(t) \subseteq \mathrm{DECLS}(\Sigma_1)$
$\Sigma' = \Sigma_1, \alpha := \lambda^n.(t[\vec{x} \mapsto n-1, ..., 0]) : A, \Sigma_2$
$\vec{x}$ *is a list of* n *distinct variables,* $\mathrm{FV}(t) \subseteq \vec{x}$
$\Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{\{x_1,...,x_n\}\cup\mathrm{FV}(B_1),\mathrm{FV}(t)\cup\mathrm{FV}(B_2)} \Gamma_2$ *and*
$\Sigma;\Gamma_1\ddagger\Gamma_2 \vdash B_1 \cong\{B\}\cong B_2 : \mathrm{Set}\ddagger\mathrm{Set}$

Metavariable Argument Killing and Simplification

The following rule, which is based on Abel and Pientka's rule "Eliminating projections" [2], can be used to deal with metavariable arguments such as $x\,.\pi_1$. (The rule is presented using named variables rather than de Bruijn indices in order to make it easier to read.)

Rule Schema 12 (Context variable currying)**.** $\Sigma;\Gamma_1\ddagger\Gamma_2, x : \Sigma U_1 V_1 \ddagger \Sigma U_2 V_2, \Delta_1\ddagger\Delta_2 \vdash t_1 \approx t_2 : A_1\ddagger A_2 \rightsquigarrow \Sigma;\Gamma_1\ddagger\Gamma_2, x_1 : U_1\ddagger U_2, x_2 : V_1\ddagger V_2, \Delta_1'\ddagger\Delta_2' \vdash t_1' \approx t_2' : A_1'\ddagger A_2'$
where $\Delta_1' = \Delta_1[\langle x_1, x_2\rangle/x]$, $t_1' = t_1[\langle x_1, x_2\rangle/x]$, $A_1' = A_1[\langle x_1, x_2\rangle/x]$, $\Delta_2' = \Delta_2[\langle x_1, x_2\rangle/x]$, $t_2' = t_2[\langle x_1, x_2\rangle/x]$, $A_2' = A_2[\langle x_1, x_2\rangle/x]$

Other rules proposed by Abel and Pientka [2], or Gundry and McBride [9, 10], can be used to remove or modify arguments from a metavariable application which might make it possible to make progress on some constraints. These rules are called pruning [2, 9], same metavariable unification [2] (a.k.a. solving flex-flex equations by intersection [9]) or flattening of Σ-types [2] (a.k.a. metavariable simplification [9]). We also use these rules in the implementation.

3.4 Stating A Correctness Property

In this section we state correctness properties for rule schemas, and for the overall unification algorithm. We also sketch proofs for the soundness of some of the rule schemas.

Definition 3.12 (Constraint satisfaction)**.** Given an internal problem $\Sigma;\vec{\mathcal{C}}$ we say that $\vec{\mathcal{C}}$ is satisfied in Σ (written $\Sigma \models_{\approx} \vec{\mathcal{C}}$) if, for each constraint $\mathcal{C} = \Gamma\ddagger\Gamma' \vdash t \cong^? u : A\ddagger A' \in \vec{\mathcal{C}}$, we have $\Sigma;\Gamma\ddagger\Gamma' \vdash t \cong u : A\ddagger A'$.

Definition 3.13 (Soundness of a rule schema)**.** A rule schema is sound if, for each rule $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$ in the schema such that $\Sigma;\vec{\mathcal{C}}$ is a well-formed internal problem, we have that (i) $\Sigma \sqsubseteq \Sigma'$ (in particular, Σ' **sig**), (ii) $\Sigma';\vec{\mathcal{D}}$ is a well-formed internal problem, and (iii) for every Σ'' with $\Sigma'' \sqsupseteq \Sigma'$, if $\Sigma'' \models_{\approx} \vec{\mathcal{D}}$ then $\Sigma'' \models_{\approx} \vec{\mathcal{C}}$.

One of the aims of the definition of heterogeneous equality (Definition 3.6) is to make Rule Schema 1 sound.

Lemma 3.14. *Rule Schema 1 is sound.*

Proof. By the premises of Definition 3.13, $\Sigma;\vec{\mathcal{C}}$ is a well-formed internal problem. By Definition 2.3, Σ **sig**, $\Sigma;\Gamma \vdash t : A$ and $\Sigma;\Gamma' \vdash t : A'$.

(i) By Definition 3.9, $\Sigma \sqsubseteq \Sigma$.

(ii) We have Σ **sig**, and the problem $\Sigma;\square$ has no constraints, so by Definition 2.3 the problem is well-formed.

(iii) Assume $\Sigma'' \sqsupseteq \Sigma$. By Statement 2 we have $\Sigma'';\Gamma \vdash t : A$ and $\Sigma'';\Gamma' \vdash t \equiv t : A'$. By reflexivity of equality (see Definition 3.5) $\Sigma'';\Gamma \vdash t \equiv t : A$ and $\Sigma'';\Gamma' \vdash t \equiv t : A'$. By Definition 3.6 we get that $\Sigma'';\Gamma \ddagger \Gamma' \vdash t \cong\{t\}\cong t : A \ddagger A'$ (because $\textsc{fv}(t) \subseteq \textsc{fv}(t) \cap \textsc{fv}(t)$). By Definition 3.12 we can conclude that $\Sigma'' \models\approx \mathcal{C}$. ◻

Statement 5 (λ inversion)**.** *If* $\Sigma;\Gamma \vdash \lambda.t : \Pi A B$, *then* $\Sigma;\Gamma, A \vdash t : B$.

Lemma 3.15. *Rule Schema 2 is sound.*
Proof sketch.

1. By the assumption that $\Sigma;\vec{\mathcal{C}}$ is well-formed we have Σ **sig**, and thus (by Definition 3.9) $\Sigma \sqsubseteq \Sigma$.
2. By the same assumption $\Sigma;\Gamma \vdash \lambda.t : \Pi A B$ and $\Sigma;\Gamma' \vdash \lambda.u : \Pi A' B'$, which by Statement 5 gives $\Sigma;\Gamma, A \vdash t : B$ and $\Sigma;\Gamma', A' \vdash u : B'$. Because Σ **sig** we get that $\Sigma;\vec{\mathcal{D}}$ is well-formed.
3. If $\Sigma'';\Gamma \ddagger \Gamma', A \ddagger A' \vdash t \cong\{v\}\cong u : B \ddagger B'$ then $\Sigma'';\Gamma \ddagger \Gamma' \vdash \lambda.t \cong\{\lambda.v\}\cong \lambda.u : \Pi A B \ddagger \Pi A' B'$ (use the $\textsc{abs}$ rule twice). ◻

In §2.3, the core of the issue is a metavariable being instantiated to a term of a mismatched type. The constraints placed on the metavariable instantiation rule preclude this.

Statement 6 (Signature strengthening)**.** *If* $\textsc{consts}(\Gamma) \cup \textsc{consts}(t) \cup \textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$, *and* $\Sigma;\Gamma \vdash t : A$ *with* $\Sigma = \Sigma_1, \Sigma_2$ *for some* Σ_2, *then* $\Sigma_1;\Gamma \vdash t : A$. *This property generalizes to other judgments.*

Statement 7 (No extraneous constants)**.** *For any* Σ, Γ *and* A, *if* $\Sigma;\Gamma \vdash A$ **type**, *then* $\textsc{consts}(\Gamma) \cup \textsc{consts}(A) \subseteq \textsc{decls}(\Sigma)$. *This property generalizes to other judgments; in particular, if* $\Sigma_1, \alpha : A, \Sigma_2$ **sig** *then* $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$.

Statement 8 (Typing of abstractions)**.** *If* $\alpha : A \in \Sigma$, $\Sigma;\Gamma \vdash \alpha\,\vec{x}^n : B$, *where all the variables in the vector* $\vec{x}$ *are distinct,* $\Sigma;\Gamma \vdash t : B$, *and* $\textsc{fv}(t) \subseteq \{\vec{x}\}$, *then* $\Sigma;\cdot \vdash \lambda^n.(t[\vec{x} \mapsto n-1, ..., 0]) : A$ *and* $\lambda^n.(t[\vec{x} \mapsto n-1, ..., 0])\ @\ \vec{x} = t$.

Statement 9 (Replacement of neutrals)**.** *If* $\Sigma;\Gamma \vdash h \Rightarrow A$, $\Sigma;\Gamma \vdash t : A$ *and* $\Sigma;\Gamma \vdash h\,\vec{e} : T$, *then* $\Sigma;\Gamma \vdash t\ @\ \vec{e} : T$.

Statement 10 (Context weakening)**.** *If* $\Sigma;\cdot \vdash t : A$ *and* $\Sigma \vdash \Gamma$ **ctx**, *then* $\Sigma;\Gamma \vdash t : A$.

Lemma 3.16. *Rule Schema 11 is sound.*
Proof sketch.

(i) Because the original problem is well-formed we have Σ **sig**, $\Sigma;\Gamma_1 \vdash \alpha\,\vec{x} : B_1$ and $\Sigma;\Gamma_2 \vdash t : B_2$. The rule preconditions and Lemma 3.11 imply that $\Sigma;\Gamma \vdash \alpha\,\vec{x} : B$ and $\Sigma;\Gamma \vdash t : B$. Let $t' = t[\vec{x} \mapsto n-1, ..., 0]$. By Statement 8 we have $\Sigma;\cdot \vdash \lambda^n.t' : A$. By Statement 7, $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$. Statement 6 implies that $\Sigma_1;\cdot \vdash \lambda^n.t' : A$, so Σ' **sig**. By Definition 3.9 we get that $\Sigma \sqsubseteq \Sigma'$.

(ii) Because Σ' **sig**, $\Sigma';\square$ is a well-formed problem.

(iii) Assume $\Sigma'' \sqsupseteq \Sigma'$.
By Statement 8 we get that $\Sigma;\cdot \vdash \lambda^n.t' : A$ (as mentioned above) and $\lambda^n.t'\ @\ \vec{x} = t$. By Lemma 3.4, $\Sigma' \vdash \Gamma_1$ **ctx**. Statement 10 and Statement 2, we have $\Sigma';\Gamma_1 \vdash \lambda^n.t' : A$. By rule $\textsc{meta}_2$, $\Sigma';\Gamma_1 \vdash \alpha \Rightarrow A$. By Statement 2 and Statement 9, $\Sigma';\Gamma_1 \vdash t : B_1$. Thus, by rule $\textsc{delta-meta}$ and Statement 2, $\Sigma';\Gamma_1 \vdash \alpha\,\vec{x} \equiv t : B_1$. By Statement 2, $\Sigma'';\Gamma_1 \vdash \alpha\,\vec{x} \equiv t : B_1$.
By reflexivity and Statement 2 we get that $\Sigma'';\Gamma_2 \vdash t \equiv t : B_2$. Because $\textsc{fv}(t) \subseteq \vec{x}$ we have $\textsc{fv}(t) \subseteq \textsc{fv}(\alpha\,\vec{x}) \cap \textsc{fv}(t)$. By Definition 3.12 we can conclude that $\Sigma'' \models\approx \vec{\mathcal{C}}$. ◻

A solution to an internal unification problem may be produced by chaining correct unification rules together.

Definition 3.17 (Problem reduction)**.** We say that the problem $\Sigma;\vec{\mathcal{E}}$ reduces to $\Sigma';\vec{\mathcal{E}'}$ in one step (written $\Sigma;\vec{\mathcal{E}} \rightsquigarrow \Sigma';\vec{\mathcal{E}'}$), if $\vec{\mathcal{E}} = \vec{\mathcal{E}_1} \wedge \vec{\mathcal{C}} \wedge \vec{\mathcal{E}_2}$, $\vec{\mathcal{E}'} = \vec{\mathcal{E}_1} \wedge \vec{\mathcal{D}} \wedge \vec{\mathcal{E}_2}$, and $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$ is a correct rule. We say that the problem $\Sigma;\vec{\mathcal{E}}$ reduces to $\Sigma';\vec{\mathcal{E}'}$ if $\Sigma;\vec{\mathcal{E}} \rightsquigarrow^\star \Sigma';\vec{\mathcal{E}'}$, where $_ \rightsquigarrow^\star _$ is the reflexive, transitive closure of $_ \rightsquigarrow _$.

Chaining sound rules results in well-typed signatures, thus avoiding the issues detailed in §2.3.

Lemma 3.18 (Soundness of unification)**.** *If* $\Sigma;\vec{\mathcal{C}} \rightsquigarrow^\star \Sigma';\square$, *where* $\Sigma;\vec{\mathcal{C}}$ *is well-formed and each step in the sequence is a sound rule, then* Σ' **sig**, $\Sigma \sqsubseteq \Sigma'$ *and* $\Sigma' \models\approx \mathcal{C}$.

Proof. By induction on the length of the sequence, using the soundness of each rule. ◻

We want our rules not only to produce well-typed solutions, but also to preserve all the possible solutions of the problem to which they are applied. We call this property completeness. This property implies that if the resulting problem has a unique solution, or no solution, then the same holds for the original problem.

With the aim of making the statement of what it means for a rule schema to be complete easier to understand, we define a notion of signature called metasubstitution in which all metavariables are instantiated.

Definition 3.19 (Metasubstitution)**.** A metasubstitution Θ is a signature of the form $\Theta ::= \cdot \mid \Theta, \mathbb{a} : A \mid \alpha := t : A$, where the types and terms in Θ are all metavariable-free.

We say that Θ is a well-formed metasubstitution (Θ **wf**) iff Θ is a metasubstitution and Θ **sig**.

Equality of metasubstitutions is defined analogously to signature extension:

Definition 3.20 (Equality of metasubstitutions)**.** We say that the metasubstitutions Θ and Θ' are equal (written $\Theta \equiv \Theta'$) if Θ_1 **wf**, Θ_2 **wf** and $\Theta_1 \equiv' \Theta_2$, where:

i) $\Theta_1, \mathbb{a} : A, \Theta_2 \equiv' \Theta_1, \mathbb{a} : A', \Theta_2$ **if** $\Theta_1; \cdot \vdash A \equiv A'$ **type**
ii) $\Theta_1, \alpha := t : A, \Theta_2 \equiv' \Theta_1, \alpha := t : A', \Theta_2$ **if** $\Theta_1; \cdot \vdash A \equiv A'$ **type**
iii) $\Theta_1, \alpha := t : A, \Theta_2 \equiv' \Theta_1, \alpha := t' : A, \Theta_2$ **if** $\Theta_1; \cdot \vdash t \equiv t' : A$
iv) $\Theta \equiv' \Theta'$ **if** Θ' is a reordering of Θ
v) $\Theta_1 \equiv' \Theta_3$ **if** $\Theta_1 \equiv \Theta_2$ **and** $\Theta_2 \equiv \Theta_3$

Definition 3.21 (Metasubstitution compatibility)**.** A metasubstitution Θ is compatible with an internal problem $\Sigma; \vec{\mathcal{C}}$ ($\Theta \vDash \Sigma; \vec{\mathcal{C}}$) if Θ and $\Sigma; \vec{\mathcal{C}}$ are well-formed and:

i) $\Theta \vDash \Sigma$, that is, $\textsc{decls}(\Theta) = \textsc{decls}(\Sigma)$ and for each declaration $D \in \Sigma$, either (i) $D = \mathbb{a} : A$, and $\Theta; \cdot \vdash \mathbb{a} : A$, or (ii) $D = \alpha : A$, and $\Theta; \cdot \vdash \alpha : A$, or (iii) $D = \alpha := u : A$, and $\Theta; \cdot \vdash \alpha \equiv u : A$.
ii) For each constraint $\mathcal{C} = \Gamma \ddagger \Gamma' \vdash t \cong^? u : A \ddagger A' \in \vec{\mathcal{C}}$, $\Theta \vDash \mathcal{C}$, that is, we have $\Theta \vdash \Gamma \equiv \Gamma'$ **ctx**, $\Theta; \Gamma \vdash A \equiv A'$ **type**, and $\Theta; \Gamma \vdash t \equiv u : A$.

We consider a rule schema *complete* if it preserves the set of solutions of the problem when applied.

Definition 3.22 (Completeness of a rule schema)**.** A rule schema is complete if, for each rule $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$ and each metasubstitution Θ such that $\Theta \vDash \Sigma;\vec{\mathcal{C}}$, there is a metasubstitution Θ' such that $\Theta' \vDash \Sigma';\vec{\mathcal{D}}$ and $\Theta = \Theta'_\Sigma$ (note the use of $=$ rather than $\equiv$).

The syntax Θ_Σ ("Θ restricted to Σ") denotes a metasubstitution which contains the same declarations as Θ, in the same order, except that declarations of metavariables that are not declared in Σ are omitted.

We now state the main correctness property (for a proof, see the first author's licentiate thesis [**?**]):

Statement 11 (Correctness of unification)**.** *Let $\Sigma; \vec{\mathcal{C}}$ be an internal problem derived from well-formed constraints of the form $\Sigma; \Gamma \vdash t : A \equiv^? u : B$ as per §2.5. Assume that $\Sigma; \vec{\mathcal{C}} \rightsquigarrow^\star \Sigma'; \square$, where Σ' has no uninstantiated metavariable declarations. Then Σ' **sig**, $\Sigma' \mathrel{|\!\!\approx} \vec{\mathcal{C}}$, and there exists a unique solution to $\Sigma;\vec{\mathcal{C}}$; that is, (i) a metasubstitution Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$, and (ii) for every $\tilde{\Theta}$ such that $\tilde{\Theta} \vDash \Sigma; \vec{\mathcal{C}}$, we have $\Theta \equiv \tilde{\Theta}$.*

3.5 Unification Example

Here we show how an algorithm may apply the rules from §3.3 in order to solve the unification problem described in §2.5, thus fulfiling our second goal (§2.4). Note how every intermediate signature and constraint is well-formed.

Example 3.23. As in §2.5, define the problem $\Sigma; \mathcal{C}_1, \Sigma; \mathcal{C}_2$ as follows:

$$\Sigma \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Bool} \to \mathrm{Set}, \alpha : \mathrm{Bool} \to \mathrm{BoolOp}$$
$$\mathcal{C}_1 \stackrel{\text{def}}{=} x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \mathbb{F}\ (\mathrm{get}\ (\alpha\ x)) \to \mathrm{BoolOp} \cong^? \mathbb{F}\ \mathrm{true} \to \mathrm{BoolOp} : \mathrm{Set}\ddagger\mathrm{Set}$$
$$\mathcal{C}_2 \stackrel{\text{def}}{=} x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \lambda y.\mathrm{None} \cong^? \lambda y.(\alpha\ x) : (\mathbb{F}\ (\mathrm{get}\ (\alpha\ x)) \to \mathrm{BoolOp})\ddagger(\mathbb{F}\ \mathrm{true} \to \mathrm{BoolOp})$$

Step 1. By applying Rule Schema 2 to $\mathcal{C}_2$, we have $\Sigma; \mathcal{C}_1, \mathcal{C}_2 \rightsquigarrow \Sigma; \mathcal{C}_1, \mathcal{C}'_2$, where $\mathcal{C}'_2 \stackrel{\text{def}}{=} x : \mathrm{Bool}\ddagger\mathrm{Bool}, y : \mathbb{F}\ (\mathrm{get}\ (\alpha\ x)) \ddagger \mathbb{F}\ \mathrm{true} \vdash \mathrm{None} \cong^? \alpha\ x : \mathrm{BoolOp}\ddagger\mathrm{BoolOp}$.
Step 2. By applying a symmetric variant of Rule Schema 11 to $\mathcal{C}'_2$, we have $\Sigma; \mathcal{C}_1, \mathcal{C}'_2 \rightsquigarrow \Sigma'; \mathcal{C}_1$, where $\Sigma' \stackrel{\text{def}}{=} \mathbb{F} : \mathrm{Bool} \to \mathrm{Set}, \alpha := \lambda y.\mathrm{None} : \mathrm{Bool} \to \mathrm{BoolOp}$.
Step 3. By applying Rule Schema 3 to $\mathcal{C}_1$, we have $\Sigma'; \mathcal{C}_1 \rightsquigarrow \Sigma'; \mathcal{C}'_1$, where $\mathcal{C}'_1 \stackrel{\text{def}}{=} x : \mathrm{Bool}\ddagger\mathrm{Bool} \vdash \mathbb{F}\ \mathrm{true} \to \mathrm{BoolOp} \cong^? \mathbb{F}\ \mathrm{true} \to \mathrm{BoolOp} : \mathrm{Set}\ddagger\mathrm{Set}$.
Step 4. By Rule-Schema 1, $\Sigma'; \mathcal{C}'_1 \rightsquigarrow \Sigma'; \square$.

Thus we have $\Sigma; \mathcal{C}_1, \mathcal{C}_2 \rightsquigarrow^\star \Sigma'; \square$. Note that Σ' is a metasubstitution satisfying $\Sigma' \vDash \Sigma'; \mathcal{C}_1, \mathcal{C}_2$. If Statement 11 holds, then this solution is unique (up to $\equiv$). ◻

4 Evaluation

To assess the practicality of the unification rules in §3 we have implemented them in a type checker. (No guarantees are made that the implementation is free of bugs.) The type checker's performance has been investigated for a small number of examples. Note that the type-checker's implementation was tuned based on these examples: we make no guarantees that it performs well in other cases.

4.1 Implementation

The type checker Tog [19] was used as a starting point. Tog implements an Agda-like language with Π-types, records with η-equality (also for the unit type), inductive-recursive data types and an identity type.

Tog deals with the pitfalls arising from Example 2.1 by always unifying types before terms, which prevents it from handling Example 2.2 successfully. We have extended Tog's constraint-solving algorithm with the rules discussed in §3.3. The resulting implementation is called Tog$^+$. Source code and usage instructions are available for download [1].

Modifications to the Theory Tog and Tog$^+$ differ from the framework described in §3 in some significant ways:

Singleton Types With η-Equality Unit types with η-equality are tricky to implement correctly. Failure to take the effects of the η rule into account may result in lack of completeness [15]. In an attempt to address this we i) restrict pruning so that it is not applied to terms of potentially singleton type and ii) avoid usage of Rule Schema 9 until it is known that the types of both sides are not singletons.

Recursive Definitions Tog admits general recursion and negative data types. We ignore issues related to this.

Unordered Signatures Signatures are ordered (Definition 3.1). For efficiency and ease of implementation, Agda, Tog and Tog$^+$ use unordered signatures. This introduces a possibility of cyclic dependencies between declarations. We use an occurs check to ensure that a metavariable is not used, directly or indirectly, in its own body. However, we do not check that a metavariable is not used in its own type. Neither does Agda [5], although such a check could be implemented.

Performance Optimizations The implementation uses a number of optimizations, including the following ones:

(i) We keep track of which constraints would result in both sides of some twin type being equal, and use this information to, for instance, check the heterogeneous context equality precondition when instantiating metavariables (Rule Schema 11).
(ii) We use an "unblocker" mechanism, extending the one implemented in Tog [19]. This mechanism keeps track of which metavariables and constraints are preventing a constraint from being reduced, and postpones work on the constraint until the blocking metavariables have been instantiated and the blocking constraints have been solved.
(iii) We use hash-consing and memoization to (at least in some cases) speed up common operations on terms and reduce memory usage. In particular, Rule Schema 1 becomes a constant time operation.

4.2 Benchmarks

We have benchmarked the implementation using an example based on one of McBride's embeddings of dependent type theory in itself [20]. We defined a small type theory inside Tog$^+$ (and Agda) using this technique, and then we defined several types inside this type theory: partial definitions of "setoid" and "precategory", and full definitions of other mathematical structures. Our code makes use of implicit arguments, and some of the constraints produced by Tog$^+$ are similar to those in Example 2.2.

The left column of Figure 3 shows the time required to type check the examples in each of the tested implementations (real execution time). We first observe that, if we compare Tog$^+$with and without the syntactic equality rule (Rule Schema 1), we observe an increase of execution time across all examples. When the syntactic equality rule is disabled, we instead have the specific cases such as $\Sigma; \Gamma \vdash \text{Set} \approx \text{Set} : \text{Set}\ddagger\text{Set}$, $\Sigma; \Gamma \vdash \text{Bool} \approx \text{Bool} : \text{Set}\ddagger\text{Set}$, $\Sigma; \Gamma \vdash \text{true} \approx \text{true} : \text{Bool}\ddagger\text{Bool}$, and $\Sigma; \Gamma \vdash \text{false} \approx \text{false} : \text{Bool}\ddagger\text{Bool}$.

Tog$^+$ uses less time than Agda for most of the examples. However, we are cautious about claiming a performance improvement with respect to Agda, because Agda does certain things that our prototype does not (even though we did turn off Agda's termination and positivity checkers). Note also that the examples were chosen because they were challenging for a previous version of Agda. Our takeaway is that, for these examples, the execution times are comparable.

The right column of Figure 3 shows the peak amount of memory used for each implementation. Tog$^+$ uses much less memory than Agda, perhaps due to the use of hash-consing. This large difference in memory usage suggests that hash-consing may be a valuable optimization in some cases.

5 Related Work

We review the history of higher-order unification in the context of dependent types, and then discuss the approaches taken by some popular implementations.

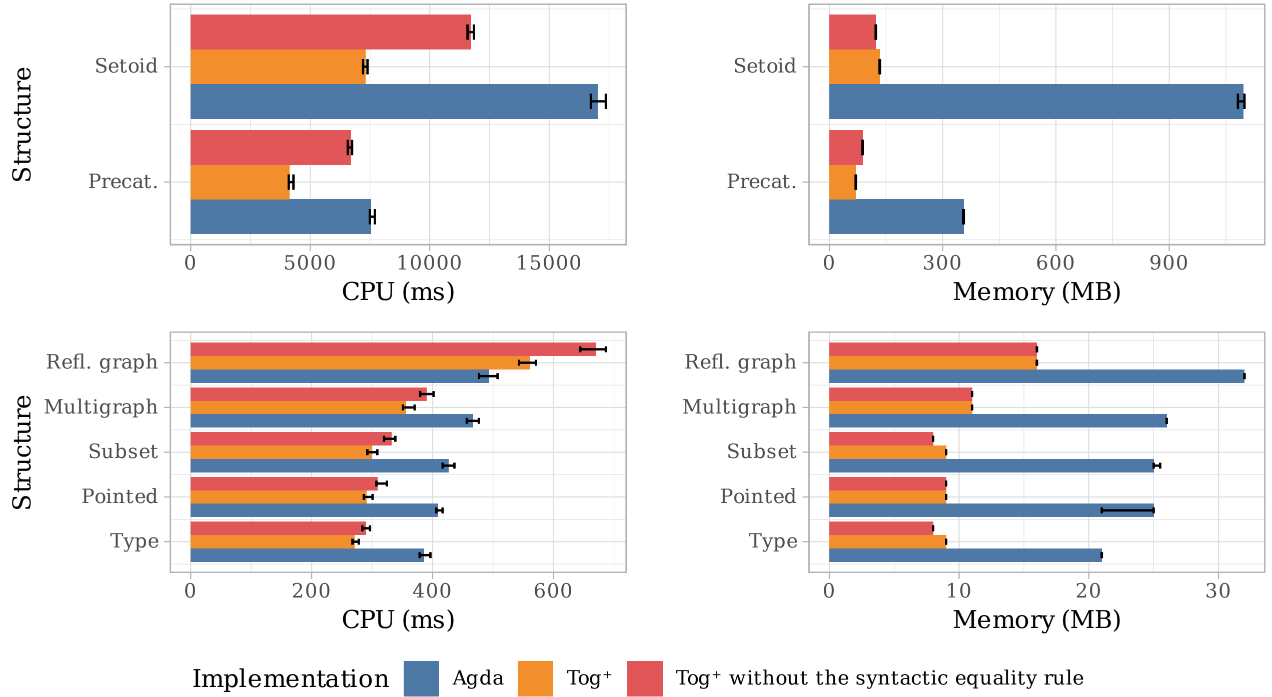
Higher-Order Unification The problem of higher-order unification is in general undecidable [11]. An algorithm for higher-order unification was proposed by Huet [12], which eventually enumerates all possible unifiers (although, due to the inherent undecidability of the problem, the algorithm may never terminate). Miller [21] discovered that, when the constraints are of a specific form (the pattern fragment), the problem becomes decidable. Muñoz [22] applied pattern unification to the λΠ calculus, and provides a calculus where the well-typedness of each step can be checked. Reed [26] showed a terminating algorithm for dynamic pattern unification. Abel and Pientka [2] extended dynamic pattern unification to handle Σ-types and η-equality.

Higher-order unification for dependent type checking with metavariables was implemented in ALF [16, 17]. In ALF terms are well-typed only modulo a set of constraints. As Norell and Coquand [23, 24] point out, ill-typed terms may cause the type-checker to loop. Their solution is to replace possibly ill-typed subterms by guarded constants; that is, terms which are blocked from normalizing until a given constraint is satisfied.

Uniqueness and the open-world assumption Implementations of algorithms performing logical reasoning may or may not conform to the open-world assumption (OWA), which is "the assumption that what is not known to be true or false might be true" [13].

Under the OWA, given the signature $\Sigma = \mathbb{A} : \text{Set}, \mathbb{c} : \mathbb{A}$, whether "$t = \mathbb{c}$ is the unique term such that $\Sigma; \cdot \vdash t : \mathbb{A}$" is unknown, as this will not hold if Σ is extended with, for

Figure 3. Median resource usage for the TT-in-TT examples. The top row shows the resource usage for the two largest examples, while the bottom row shows corresponding information for the rest, using different scales. Each example is benchmarked 40 times. We plot the median value and error bars spanning 95% confidence intervals (bootstrapped, 1000 samples).



example, the declaration $\mathbb{b} : \mathbb{A}$. Whether a rule respects the OWA is defined as follows:

Definition 5.1. A rule schema respects the open-world assumption if, for each rule $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$, and any declaration $D = \mathbb{c} : A$, $D = \alpha : A$, or $D = \alpha : A$ with Σ, D **sig** and Σ', D **sig**, it contains the rule $\Sigma, D;\vec{\mathcal{C}} \rightsquigarrow \Sigma', D;\vec{\mathcal{D}}$.

The metasubstitutions we define (Definition 3.19) contain no uninstantiated metavariables, similarly to the grounding metasubstitutions defined by Abel and Pientka [2]. We do not believe that the resulting notion of completeness (Definition 3.22) necessarily entails the open-world assumption. However, it holds for the rules defined in §3.3.

Lemma 5.2. *The rule schemas defined in §3.3 respect the open-world assumption.*

Proof sketch. Let $\Sigma;\vec{\mathcal{C}} \rightsquigarrow \Sigma';\vec{\mathcal{D}}$ belong to one of the rule schemas in §3.3, and D be such that Σ, D **sig** and Σ', D **sig**. By definition of the rule schema (and, for Rule Schemas 3, 7, 9, 10 and 11, by Statement 2), then $\Sigma, D;\vec{\mathcal{C}} \rightsquigarrow \Sigma', D;\vec{\mathcal{D}}$ belongs to the same rule schema. (For Rule Schema 10 one can show by induction on the derivation, using Statement 2, that for any Σ, Σ' and D, if $\Sigma \sqsubseteq \Sigma'$, Σ, D **sig** and Σ', D **sig**, then $\Sigma, D \sqsubseteq \Sigma', D$.) □

An alternative approach, taken by Gundry [9], is to consider solutions with uninstantiated metavariables, and use a notion of most general metasubstitutions instead of uniqueness.

Agda Agda uses dynamic pattern unification with postponement [23]. That is, constraints are solved immediately if they are in the pattern fragment, and are otherwise postponed until instantiations of metavariables allow these postponed constraints to be simplified. Metavariables are only instantiated when the solution is unique. Our prototype uses the same technique. However, in Agda constraints have a single context and a single type. Agda instead uses guarded constants, which, as shown by Norell and Coquand [24], can be effective in avoiding looping behaviour in the type-checker.

When subjected to the constraints in Example 2.1 Agda replaces the type of x with a guarded constant p of type Set, yielding the constraint $x : p \vdash \alpha\, x \equiv^{?} \mathbb{D}\,(f\,(\beta\,0)\,x) : \mathrm{Set}$. (The constant p normalizes to $\mathbb{N}$ only if $\cdot \vdash \mathbb{N} \equiv F\,(\beta\,0) : \mathrm{Set}$.) Despite guarding the type of x, the type checker still performs the ill-typed instantiation that we discuss in §2.3.

In the course of this work the first author discovered that this issue [25] can be solved using further application of the technique of guarded constants, and proposed a fix to the Agda developers. However, this fix was reverted after the second author found that it caused a regression [7].

For another open issue [6] we are unaware of any fix not involving a heterogeneous approach to unification such as the one discussed in this paper (but that does not mean that there are none).

Finally, it is not clear how to handle the unit type with η-equality while preserving completess. Agda renounces completeness in some cases [15], while we opt to implement a weaker notion of pruning with the aim of avoiding the issue.

Other Proof Assistants Other languages/proof assistants based on intensional type theory include Coq [28], Idris [4], Lean [8] and Matita [3]. All of them handle code that we created based on Example 2.1 adequately, but they fail to type check code that we created based on Example 2.2.

Gundry and McBride As mentioned above the approach used in this text is based on the one by Gundry and McBride [9, 10]. They use a theory with twin types, twin variables, a ternary notion of judgmental equality, and two universe levels, where Π-types and Σ-types may only be formed with types of the first level. They prove that their system only produces well-typed, most general solutions.

For simplicity we use a theory with a type-in-type rule. This allows us to support complex examples such as the one we benchmark (§4.2), which uses terms of the form $\Pi \mathrm{Set} B$ or $\Sigma A \mathrm{Set}$ which cannot be typed in their theory. We justify the well-typedness of our rules under certain assumptions that we hope would hold in a properly stratified version of the theory.

Gundry and McBride's judgmental equality is ternary, of the form $\Theta | \Gamma \vdash A \ni t \equiv\!\![v]\!\!\equiv u$, which is read as t is equal to u at type A in context Γ and metacontext Θ. The metacontext Θ may contain metavariable declarations ($\alpha : A$) and instantiations ($\alpha := t : A$), as well as unification constraints. The context Γ may contain both single-typed ($x : T$) and twin-typed ($x : T_1 \ddagger T_2$) variables. The witness v is in $\beta\eta\delta$-normal form at type A.

The heterogeneous equality used in this work (Definition 3.6) is inspired by Gundry and McBride's ternary equality. However, there are some differences:

(i) Our heterogeneous equality is distinct from the judgmental equality. The idea is that it should be possible to use the methods described in this text in the implementation of a language like Agda without having to switch to a new form of judgmental equality.

(ii) Our heterogeneous equality can equate terms of different types. This enables the implementation of Rule Schema 1 as it is, without any need to, say, check that the types are equal.

Our implementation generalizes the approach to support inductive-recursive types and parameterized records with η-equality, including a unit type with η, with the corresponding adaptations and generalizations of the unification rules. This allowed us to test the implementation with a complex example (§4.2). (However, we have not proved that these extensions are implemented correctly.)

Gundry and McBride's approach is formulated with fully βδ-normalized terms (i.e. with all instantiated metavariables immediately replaced by their bodies), and this simplifies some things. We have chosen to use terms in β-normal form but allow δ-redexes to remain in the terms, with the aim of keeping the theory close to existing implementations such as Agda. Note that overly eager normalisation of terms can have adverse effects on performance.

Gundry also gives criteria to detect, in some cases, whether a constraint is unsolvable, which we do not address.

Rule Schema 12 in this text supports the case where the curried variable (e.g. x) has a twin type (i.e. $x : \Sigma A_1 B_1 \ddagger \Sigma A_2 B_2$). This contrasts with the rule in the original twin approach [9, expression 4.22] in which the variable must have a single type, but instead, the type may be of the form $x : \Pi A_1 ... \Pi A_n \Sigma U_1 U_2$. We believe that this difference is not fundamental, presumably both approaches could be updated to support both features (i.e. $x : \Pi A_1 ... \Pi A_n \Sigma U_1 U_2 \ddagger \Pi B_1 ... \Pi B_n \Sigma V_1 V_2$).

6 Conclusions

We have presented an approach to higher-order unification with dependent types. Under suitable assumptions that we hope would hold in a fully stratified version of the theory, the solutions produced are well-typed (see Lemma 3.18), fulfilling Goal #1 (§2.3). The approach allows terms to be (at least partially) unified before their types are known to be (fully) equal (see §3.5), fulfilling Goal #2 (§2.4).

Our approach is based on that of Gundry and McBride [9, 10], but with a type theory more similar to the one used in Agda's implementation. We imagine that this could make it a little easier to use the approach in an existing type checker.

We have tested this approach by implementing the unification rules in an existing type checker for a tiny variant of Agda. The benchmarks show that the resulting type checker is able to handle some challenging examples which Coq, Idris, Lean and Matita cannot, and that it does so in time and space comparable to that used by Agda. We are cautious about claiming a performance improvement with respect to Agda, but we see the reduction in resource usage in our case study (especially space usage) as promising.

Acknowledgments

We want to thank Andreas Abel, Jesper Cockx, Ulf Norell and Andrea Vezzosi for discussions on the subject of higher-order unification with dependent types, and its implementation in Agda. We also want to thank Adam Gundry for clarifications about his PhD thesis, which was one of the starting points for this work. Finally, we thank the reviewers of this paper and a previous version thereof for their useful feedback.

References

[1] Andreas Abel, Nils Anders Danielsson, Francesco Mazzoli, Víctor López Juan, Andrea Vezzosi, et al. 2020. Tog^{+}. https://lopezjuan.com/project/togt/

[2] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *TLCA 2011*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21691-6_5

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita interactive theorem prover. In *International Conference on Automated Deduction*. Springer, 64–69. https://doi.org/10.1.1.229.4032

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[5] Jesper Cockx, Nils Anders Danielsson, and Andreas Abel. 2015. Issue #1556: Agda allows "very dependent" types. Agda Issue Tracker. https://github.com/agda/agda/issues/2876

[6] Nils Anders Danielsson. 2014. Issue 1467: Inconsistent constraints leading to violated invariants in conversion checking. Agda Issue Tracker. https://github.com/agda/agda/issues/1467

[7] Nils Anders Danielsson. 2020. Issue 4408: Regression related to fix of #3027. Agda Issue Tracker. https://github.com/agda/agda/issues/4408

[8] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. 2015. Elaboration in Dependent Type Theory. arXiv:cs.LO/1505.04324

[9] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

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

[11] Gérard P Huet. 1973. The undecidability of unification in third order logic. *Information and control* 22, 3 (1973), 257–267. https://doi.org/10.1016/0304-3975(81)90040-2

[12] Gerard P. Huet. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* 1, 1 (1975), 27–57. https://doi.org/10.1016/0304-3975(75)90011-0

[13] C. Maria Keet. 2013. Open World Assumption. In *Encyclopedia of Systems Biology*, Werner Dubitzky, Olaf Wolkenhauer, Kwang-Hyun Cho, and Hiroki Yokota (Eds.). Springer New York, 1567–1567. https://doi.org/10.1007/978-1-4419-9863-7_734

[14] Víctor López Juan. 2020. *Practical Unification for Dependent Type Checking*. Licentiate Thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. https://lopezjuan.com/project/licentiate/

[15] Víctor López Juan, Andreas Abel, Nils Anders Danielsson, Ulf Norell, and Jesper Cockx. 2017. Issue 2876: Overzealous pruning (reprise). Agda Issue Tracker. https://github.com/agda/agda/issues/2876

[16] Lena Magnusson. 1994. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. Dissertation.

[17] Lena Magnusson and Bengt Nordström. 1993. The ALF proof editor and its proof engine. In *International Workshop on Types for Proofs and Programs*. Springer, 213–237. https://doi.org/10.1007/3-540-58085-9_78

[18] Francesco Mazzoli and Andreas Abel. 2016. Type checking through unification. Preprint Arxiv 1609.09709v1. arXiv:cs.PL/1609.09709v1

[19] Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. 2017. Tog - A prototypical implementation of dependent types. https://github.com/bitonic/tog

[20] Conor McBride. 2010. Outrageous but Meaningful Coincidences: Dependent type-safe syntax and evaluation. In *WGP'10*. https://doi.org/10.1145/1863495.1863497

[21] Dale Miller. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation* 1, 4 (1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

[22] César Muñoz. 2001. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theoretical Computer Science* 266, 1-2 (2001), 407–440. https://doi.org/10.1016/S0304-3975(00)00196-1

[23] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation.

[24] Ulf Norell and Catarina Coquand. 2007. Type checking in the presence of meta-variables. (2007). http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html Unpublished.

[25] Ulf Norell and Víctor López Juan. 2018. Issue 3027: Internal error in the presence of unsatisfiable constraints. Agda Issue Tracker. https://github.com/agda/agda/issues/3027

[26] Jason Reed. 2009. Higher-order Constraint Simplification in Dependent Type Theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*. ACM, New York, NY, USA, 49–56. https://doi.org/10.1145/1577824.1577832

[27] Martin Stone Davis, Andreas Abel, and Ulf Norell. 2017. Issue 2709: (No longer an) Internal error at src/full/Agda/TypeChecking/Substitute.hs:98. Agda Issue Tracker. https://github.com/agda/agda/issues/2709

[28] The Coq Development Team. 2020. *Coq 8.11.0*. https://coq.inria.fr

[29] Beta Ziliani and Matthieu Sozeau. 2017. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796817000028