Practical dependent type checking using twin types

Citation for the original published paper (version of record):
López Juan, V., Danielsson, N. (2020). Practical dependent type checking using twin types. TyDe 2020 - Proceedings of the 5th ACM SIGPLAN International Workshop on Type-Driven Development, co-located with ICFP 2020: 11-23. http://dx.doi.org/10.1145/3406089.3409030

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.

With Gundry and McBride's twin variable technique one can also start 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 at least one example 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.

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

Keywords: type checking, unification, dependent types

TyDe '20, August 23, 2020, Virtual Event, USA

ACM Reference Format:
Víctor López Juan and Nils Anders Danielsson. 2020. Practical Dependent Type Checking using Twin Types. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '20), August 23, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3406089.3409030

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 [14]. 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 the program might not have the meaning that the programmer intended.

Some systems, for instance Coq [33], 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 [33] 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). The approach has the advantage that one does not need to know details of the unification algorithm, such as in what order different things are done, in order to understand a piece of code. Thus there is perhaps less need to make the unification algorithm predictable to users. A drawback is that sometimes "natural" solutions are not found.

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 [4, 20].

Gundry and McBride's technique with twin types and twin variables [13] also makes it possible to start unifying terms before their types are known to be equal, and furthermore Gundry presents correctness proofs [12]. 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 [24]).
- We use a reformulation of the approach of Gundry and McBride without twin variables. This might make the approach a little easier to adopt in existing type checkers that do not already use twin variables, because there might be less need to modify the grammar of terms or the algorithms that manipulate them.
- The approach can handle at least one example that is handled by Agda (2.6.1) but not by Coq (8.11.0), Idris (1.3.2), Lean (3.4.2) or Matita (0.99.3); see §2.4.
- 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 2.6.1.
- We introduce a notion of heterogeneous equality (§3.2) that enables a type-agnostic syntactic equality rule (Rule Schema 1). 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 [18]. 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 inspired by those of Agda and Haskell.

2.1 Unique Solutions

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

$$\mathit{replicate} : \{n : \mathrm{Nat}\} \to \mathrm{Int} \to \mathrm{Vec}\ n\ \mathrm{Int}$$

The first (implicit) argument to *replicate* determines the length of the resulting vector: *replicate* $\{n = 5\}$ (-4) gives $[-4, -4, -4, -4, -4]$. If the argument is not given explicitly, then an attempt is made to infer it from the context: *replicate* $(-4) : \mathrm{Vec}\ 3\ \mathrm{Int}$ gives $[-4, -4, -4]$.

In some cases it may not be possible to determine the length uniquely. Let us assume that there is a function *rotate90*, which rotates a vector 90 degrees clockwise (*rotate90* $[1, 2] = [-2, 1]$), and a command *print*, which prints a vector, and consider the program *main*:

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

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 do not want the type checker to fill in arbitrary type-correct terms: we do not want it to make unforced choices on behalf of the programmer. This applies not only to programs, but also to statements of theorems, and—in proof-relevant settings—bodies of proofs. (We do not claim that any proof assistant in use today would make an unforced choice in this particular case, this is a contrived example used to illustrate our point.)

2.2 Constraints and Solutions

In our development all judgments about terms and types are formulated with respect to a signature. A signature Σ contains atom declarations ($\mathbb{a} : A$), corresponding to Agda postulates, and metavariable declarations, where a metavariable is either just declared ($\alpha : A$), or declared and instantiated ($\alpha := t : A$):

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

Metavariables can be introduced by the type checker (omitted implicit arguments are often turned into metavariables), or by the unifier (when it "kills" metavariable arguments, see §3.3.2).

According to Mazzoli and Abel [23] 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, where all the constraints are well-typed with respect to Σ: $\Sigma; \Gamma_i \vdash t_i : A_i$ and $\Sigma; \Gamma_i \vdash u_i : B_i$.

A solution to a higher-order unification problem is an assignment of terms of appropriate types to all the uninstantiated metavariables in Σ, yielding a signature Σ' for which 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 $\Sigma'; \Gamma_i \vdash t_i \equiv u_i : A_i$). Σ' may contain additional metavariables as long as they are instantiated.

Let us now discuss two goals for our unification algorithm: we want to ensure that the algorithm can apply a wide range of techniques (for instance out of order unification), but at the same time we want to avoid producing solutions that are ill-typed.

2.3 Goal #1: Well-Typedness

As mentioned above metavariable solutions should be well-typed. However, we do not want to repeatedly check that all terms are well-typed. Instead we assume that we start with a well-formed signature and well-formed constraints, and aim to maintain well-typedness as an invariant of the unification algorithm.

The current implementation of Agda sometimes constructs ill-typed solutions, which can lead to crashes. The following example is taken from a currently open bug report [20]:

Example 2.1. Consider the following two functions:

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

Using these functions we can form the following constraints, which are well-formed with respect to the signature $\mathbb{D} : \text{Nat} \to \text{Set}$, $\alpha : \text{Nat} \to \text{Set}$, $\beta : \text{Nat} \to \text{Bool}$:

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

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

Note that Agda unifies $\alpha\ x$ and $\mathbb{D}\ (f\ (\beta\ 0)\ x)$ without knowing that the terms have the same type. Another approach is to only unify terms with equal types. However, this could be restrictive in practice, as we explain in the following section.

2.4 Goal #2: Out of Order Unification

McBride [25] describes a method for representing dependently typed languages inside dependently typed languages. We have used some programs based on this method in our benchmarks (see §4.2). Some of these programs yield unification constraints in which metavariables appear both in the term and the type. The following example is a reduced version of one of these constraints:

Example 2.2. BoolOp is a type of optional booleans, and *get* is a function that returns the boolean, if any:

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

The following constraint is well-formed with respect to the signature $\Sigma \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \alpha : \text{Bool} \to \text{BoolOp}$:

$$\begin{array}{r} x : \text{Bool} \vdash \lambda y.\text{None} : \mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp} \equiv^? \\ \lambda y.(\alpha\ x) : \mathbb{F}\ \text{true} \to \text{BoolOp} \end{array}$$ ■

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:

$$\begin{array}{l} \Sigma'; x : \text{Bool} \vdash \mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp} \equiv \\ \qquad \mathbb{F}\ \text{true} \to \text{BoolOp} : \text{Set} \qquad (1) \\ \Sigma'; x : \text{Bool} \vdash (\lambda y.\text{None}) \equiv \\ \qquad (\lambda y.(\alpha\ x)) : \mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp} \qquad (2) \end{array}$$

The first equality (1) does not contain enough information to determine t; both $t = \lambda x.\text{None}$ and $t = \lambda x.\text{Some true}$ are possible. The second equality (2) does contain enough information (the only solution is $t = \lambda x.\text{None}$), but the term $\lambda y.(\alpha\ x)$ is compared at the type $\mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp}$, which is not its original one ($\mathbb{F}\ \text{true} \to \text{BoolOp}$).

We ported this example to five different proof assistants, and found that Agda (2.6.1) instantiated α, but none of Coq (8.11.0), Idris (1.3.2), Lean (3.4.2), or Matita (0.99.3). Perhaps the proof assistants that did not instantiate α do not unify terms unless they are known to have the same types.

2.5 Twin Types

Gundry and McBride [13, 12] propose a heterogeneous approach to unification, in which constraints have two types, one for each side. They also use twin variables, variables with two types ($\hat{x} : A_1 \ddagger A_2$), where syntax is used to specify the type of a specific use of a variable ($\acute{x} : A_1$ and $\grave{x} : A_2$). We propose a variant of this approach in which twin types ($A_1 \ddagger A_2$) are only used for constraints and twin variables are not used: the type of each occurrence of a variable is instead determined by the side it occurs on. If twin types are only used for constraints, then it might be a little easier to adapt existing type checkers to use this approach.

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 ($\Gamma \ddagger \Gamma \vdash A \cong^? B : \text{Set} \ddagger \text{Set}$) 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 $\Gamma_1 ‡ \Gamma_2 \vdash t \cong^? u : A ‡ B$. The problem is well-formed if the signature is well-formed and, for each constraint $\Gamma_1 ‡ \Gamma_2 \vdash t \cong^? u : A ‡ B \in \vec{\mathcal{C}}$, 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 $\Gamma_1 = \Gamma_1', A_1$ and $\Gamma_2 = \Gamma_2', A_2$ of the same length we may write $\Gamma_1 ‡ \Gamma_2$ as $\Gamma_1' ‡ \Gamma_2', A_1 ‡ A_2$.

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:

$$\begin{aligned}
\mathcal{C}_1 &\stackrel{\text{def}}{=} x : \text{Bool} ‡ \text{Bool} \vdash \mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp} \cong^? \\
&\qquad\qquad \mathbb{F}\ \text{true} \to \text{BoolOp} : \text{Set} ‡ \text{Set} \\
\mathcal{C}_2 &\stackrel{\text{def}}{=} x : \text{Bool} ‡ \text{Bool} \vdash \lambda y.\text{None} \cong^? \lambda y.(\alpha\ x) : \\
&\qquad (\mathbb{F}\ (\mathit{get}\ (\alpha\ x)) \to \text{BoolOp}) ‡ (\mathbb{F}\ \text{true} \to \text{BoolOp})
\end{aligned}$$

All constraints are well-formed, so constraint $\mathcal{C}_2$ can be tackled immediately. If α is instantiated with $\lambda y.\text{None}$, then the two sides of the constraint become *heterogeneously* equal, as defined in §3.2. (This example is discussed further in §3.5.)

3 Dependent Type Checking using Twin Types

In this section we describe some 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.

In order to stay close to our implementation we represent variables using de Bruijn indices ($0, 1, 2, \ldots$). Indices are manipulated using weakenings ($t^{(+n)}$, increment all indices by n) and renamings ($t[\vec{x} \mapsto \vec{y}]$, replace each variable in $\vec{x}$ with the corresponding variable in $\vec{y}$, where $\vec{x}$ and $\vec{y}$ are lists of indices of the same length).

Terms are in β-normal form (following the implementation of Agda). We use hereditary substitution ($t[u/x]$) and application ($t \mathbin{@} \vec{e}$) operations that preserve the invariant that terms are in β-normal form [32, 6]. For instance, $(x\ y_1\ y_2)[\lambda z_1.\lambda z_2.z_1/x] \stackrel{\text{def}}{=} (\lambda z_1.\lambda z_2.z_1) \mathbin{@} y_1\ y_2 \stackrel{\text{def}}{=} (\lambda z_2.z_1)[y_1/z_1] \mathbin{@} y_2 \stackrel{\text{def}}{=} (\lambda z_2.y_1) \mathbin{@} y_2 \stackrel{\text{def}}{=} y_1[y_2/z_2] \stackrel{\text{def}}{=} y_2$. Hereditary substitution and application are deterministic, but are not guaranteed to terminate due to the type-in-type axiom. When we write $t[u/x]$ or $t \mathbin{@} \vec{e}$ 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)**.** The following rules define (inductively) when a signature is well-formed:

$$\frac{}{\cdot\ \mathbf{sig}}\ \textsc{empty}$$

$$\frac{\Sigma\ \mathbf{sig} \qquad \mathbb{o} \text{ is fresh in } \Sigma \qquad \Sigma; \cdot \vdash A\ \mathbf{type}}{\Sigma, \mathbb{o} : A\ \mathbf{sig}}\ \textsc{atom-decl}$$

$$\frac{\Sigma\ \mathbf{sig} \qquad \alpha \text{ is fresh in } \Sigma \qquad \Sigma; \cdot \vdash A\ \mathbf{type}}{\Sigma, \alpha : A\ \mathbf{sig}}\ \textsc{meta-decl}$$

$$\frac{\Sigma, \alpha : A\ \mathbf{sig} \qquad \Sigma; \cdot \vdash t : A}{\Sigma, \alpha := t : A\ \mathbf{sig}}\ \textsc{meta-inst}$$

The syntaxes of contexts and terms are not given explicitly, but they can be read off from the typing rules.

Definition 3.2 (Context)**.** $\Sigma \vdash \Gamma\ \mathbf{ctx}$ means that the context Γ is well-formed with respect to the signature Σ:

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

Definition 3.3 (Term and neutral term)**.** $\Sigma; \Gamma \vdash t : A$ means that the term t has type A in context Γ and signature Σ, see Figure 1. We use $t, u, \ldots, A, B, \ldots$ or f to denote terms. Terms of the form $h\ \vec{e}$, where the *head* h is of the form n, $\mathbb{o}$, α or if and each *eliminator* in $\vec{e}$ is of the form t, $.\pi_1$ or $.\pi_2$, are called neutral terms, and denoted by f.

Notation. We may use $(x : A) \to B$ or $A \to B$ instead of ΠAB, and $(x : A) \times B$ or $A \times B$ instead of ΣAB.

Definition 3.4 (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.

Statement 1 (Piecewise well-formedness of judgments)**.** *If* $\Sigma; \Gamma \vdash t : A$*, then* $\Sigma \vdash \Gamma\ \mathbf{ctx}$ *and* $\Sigma\ \mathbf{sig}$*. If* $\Sigma; \Gamma \vdash t \equiv u : A$*, then* $\Sigma; \Gamma \vdash t : A$ *and* $\Sigma; \Gamma \vdash u : A$*.*

Statement 2. *The relations* $\Sigma \vdash \_ \equiv \_\ \mathbf{ctx}$*,* $\Sigma; \Gamma \vdash \_ \equiv \_\ \mathbf{type}$ *and* $\Sigma; \Gamma \vdash \_ \equiv \_ : A$ *are reflexive, symmetric and transitive with respect to the sets* $\{\Gamma \mid \Sigma \vdash \Gamma\ \mathbf{ctx}\}$*,* $\{A \mid \Sigma; \Gamma \vdash A\ \mathbf{type}\}$ *and* $\{t \mid \Sigma; \Gamma \vdash t : A\}$*, respectively.*

Notation. We use $\textsc{fv}(t)$ to denote the set of free variables of t (e.g. $\textsc{fv}(\lambda.(3\ 0)) = \{2\}$). Similarly $\textsc{fv}(\Delta \vdash t : U)$ is used to denote the set of free variables of $\Delta \vdash t : U$ (e.g. $\textsc{fv}(\mathbb{A}, \mathbb{B}\ 3 \vdash 9\ 1 : \mathbb{C}) = \{7, 2\}$).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Judgmental equality for terms, types and contexts

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 3.5 (Heterogeneous equality)**.** If t and u are terms such that $\Sigma;\Gamma_1 \vdash t : A$ and $\Sigma;\Gamma_2 \vdash u : B$, and 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 omit v and write $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash t \cong u : A \ddagger B$.

Remark 3.6. *The heterogeneous equality is symmetric and satisfies the following form of reflexivity: given $\Sigma;\Gamma \vdash t : A$ we have $\Sigma;\Gamma \ddagger \Gamma \vdash t \cong\{t\}\cong t : A \ddagger A$.*

Remark 3.7. *If $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash t \cong\{v\}\cong u : A \ddagger B$, then by Definition 3.5 and Statement 1, we have $\Sigma;\Gamma_1 \vdash v : A$ and $\Sigma;\Gamma_2 \vdash v : B$. However, this does not mean that $\Sigma \vdash \Gamma_1 \equiv \Gamma_2\ \mathbf{ctx}$, or $\Sigma, \Gamma_1 \vdash A \equiv B\ \mathbf{type}$. For example (using variable*

names for clarity), let $\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \text{Set}, \mathbb{B} : \text{Set}$, $A \stackrel{\text{def}}{=} \mathbb{A} \to \mathbb{A}$, $B \stackrel{\text{def}}{=} \mathbb{B}$, $\Gamma_1 \stackrel{\text{def}}{=} x : B, z : A$ *and* $\Gamma_2 \stackrel{\text{def}}{=} x : A, z : B$. *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 3.5 to formulate the soundness of the rules given in the following section (Definition 3.14), 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 3.8 (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). These rules are adaptations to our setting of rules that are used in other developments.

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)**.** $\Sigma; \Gamma \ddagger \Gamma' \vdash \lambda.t \approx \lambda.u : \Pi AB \ddagger \Pi A'B' \rightsquigarrow \Sigma; \Gamma \ddagger \Gamma', A \ddagger A' \vdash t \approx u : B \ddagger B'$

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** $\text{FV}(t) \supseteq \text{FV}(t')$

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

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** $\Sigma \vdash \Gamma, A \equiv \Gamma_0, A_0$ **ctx**

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

Definition 3.9 (Strongly neutral term)**.** A neutral term f is strongly neutral if f is of the form $x\, \vec{e}$, $\mathbb{o}\, \vec{e}$, 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 boolean) is a strongly neutral term.

Rule Schema 9 (Rigid-rigid unification)**.** *If* $h\, \vec{e}$ *and* $h\, \overrightarrow{e'}$ *are strongly neutral;* $|\vec{e}| = |\overrightarrow{e'}| = n$*;* $J \subseteq \{1, ..., n\}$*; 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* $\Sigma; \Gamma' \vdash h\, \overrightarrow{e'}_{1,...,i-1} : \Pi B'_i C'_i$*, then* $\Sigma; \Gamma \ddagger \Gamma' \vdash h\, \vec{e} \approx h\, \overrightarrow{e'} : T \ddagger T' \rightsquigarrow \Sigma; \bigwedge_{i \in J} \Gamma \ddagger \Gamma' \vdash t_i \approx u_i : B_i \ddagger B'_i$

Some unification rules modify the signature by i) adding new metavariable declarations, ii) instantiating existing metavariables, iii) reordering signature entries or iv) replacing terms with convertible terms. We call the result of these operations a signature extension.

Definition 3.10 (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) $\Sigma_1, \alpha : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha := t : A, \Sigma_2$
(iii) $\Sigma_1 \sqsubseteq' \Sigma_3$ **if** $\Sigma_1 \sqsubseteq \Sigma_2$ **and** $\Sigma_2 \sqsubseteq \Sigma_3$
(iv) $\Sigma_1, \mathbb{o} : A, \Sigma_2 \sqsubseteq' \Sigma_1, \mathbb{o} : A', \Sigma_2$ **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) $\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha := t : A', \Sigma_2$ **if** $\Sigma_1; \cdot \vdash A \equiv A'$ **type**
(vii) $\Sigma_1, \alpha := t : A, \Sigma_2 \sqsubseteq' \Sigma_1, \alpha := t' : A, \Sigma_2$ **if** $\Sigma_1; \cdot \vdash t \equiv t' : A$
(viii) $\Sigma \sqsubseteq' \Sigma'$ **if** Σ' is a (possibly trivial) reordering of Σ

Definition 3.10 is intended to justify Statement 3:

Statement 3 (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.*

Reordering and normalizing signature entries is allowed:

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

3.3.1 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 [12, p. 69], but does not formalize it.)

Definition 3.11 (Heterogeneous context equality for sets of variables)**.** Given $X \subseteq \mathbb{N}$, let $X{-}1 \stackrel{\text{def}}{=} \{x{-}1 | x \in X, x > 0\}$. If the contexts Γ_1 and Γ_2 are well-formed with respect to Σ, then the following rules define when they are heterogeneously equal with respect to the signature Σ and the sets of variables X_1 and X_2, with Γ as witness (written $\Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{X_1, X_2} \Gamma_2$).

$$\frac{\Sigma\ \mathbf{sig}}{\Sigma \vdash \cdot \cong\{\,\cdot\,\}\cong_{\emptyset,\emptyset} \cdot}\ \textsc{empty}$$

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

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

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

$$\frac{0 \in X_1 \cap X_2 \qquad \Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash A_1 \cong\{A\}\cong A_2 : \mathrm{Set}\ddagger\mathrm{Set} \qquad \Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{(X_1-1)\cup \mathrm{FV}(A_1),(X_2-1)\cup \mathrm{FV}(A_2)} \Gamma_2}{\Sigma \vdash \Gamma_1, A_1 \cong\{\Gamma, A\}\cong_{X_1,X_2} \Gamma_2, A_2}\ \textsc{used}$$

Here are more statements about the theory, and a lemma:

Statement 4 (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 5 (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 3.12 (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 4 and 5 and the interpolant property from Definition 3.5. (See the first author's licentiate thesis [18] 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 n (not necessarily distinct) 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* $\textsc{consts}(t) \subseteq \textsc{decls}(\Sigma_1)$*,* $\Sigma' = \Sigma_1, \alpha := \lambda^n.(t[\vec{x}^n \mapsto n-1, ..., 0]) : A, \Sigma_2$*,* $\vec{x}^n$ *is a list of* n *distinct variables,* $\mathrm{FV}(t) \subseteq \vec{x}^n$*,* $\Sigma \vdash \Gamma_1 \cong\{\Gamma\}\cong_{\{x_1,...,x_n\}\cup\mathrm{FV}(B_1),\mathrm{FV}(t)\cup\mathrm{FV}(B_2)} \Gamma_2$ *and* $\Sigma;\Gamma_1 \ddagger \Gamma_2 \vdash B_1 \cong\{B\}\cong B_2 : \mathrm{Set}\ddagger\mathrm{Set}$

3.3.2 Other Rules.

The following rule, which is based on Abel and Pientka's rule "Eliminating projections" [5], 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). $\Sigma;\Gamma_1 \ddagger \Gamma_2, x : \Sigma x_1 : U_1.V_1 \ddagger \Sigma x_1 : U_2.V_2, \Delta_1 \ddagger \Delta_2 \vdash t_1 \approx t_2 : A_1 \ddagger A_2 \rightsquigarrow \Sigma;\Gamma_1 \ddagger \Gamma_2, x_1 : U_1 \ddagger U_2, x_2 : V_1 \ddagger V_2, \Delta_1' \ddagger \Delta_2' \vdash t_1' \approx t_2' : A_1' \ddagger A_2'$
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 [5] and Gundry and McBride [13, 12] can be used to remove ("kill") eliminators from a metavariable-headed neutral term. This might make it possible to make progress on some constraints. These rules are called pruning [5, 12], same metavariable unification [5]/solving flex-flex equations by intersection [12], and flattening of Σ-types [5]/metavariable simplification [12]. We also use rules of this kind in the implementation.

3.4 A Correctness Property

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

Definition 3.13 (Constraint satisfaction). Given an internal problem $\Sigma;\vec{\mathcal{C}}$ we say that Σ satisfies $\vec{\mathcal{C}}$ (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 3.14 (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 3.5) is to make Rule Schema 1 sound.

Lemma 3.15. *Rule Schema 1 is sound.*

Proof. By the premises of Definition 3.14, $\Sigma;\vec{\mathcal{C}}$ 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 3.10, $\Sigma \sqsubseteq \Sigma$.

(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 3 we have $\Sigma'';\Gamma \vdash t : A$ and $\Sigma'';\Gamma' \vdash t \equiv t : A'$. By reflexivity of equality (see Definition 3.4) $\Sigma'';\Gamma \vdash t \equiv t : A$ and $\Sigma'';\Gamma' \vdash$

$t \equiv t : A'$. By Definition 3.5 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 3.13 we can conclude that $\Sigma'' \not\approx \mathcal{C}$. □

Statement 6 (λ inversion). *If* $\Sigma; \Gamma \vdash \lambda.t : \Pi A B$, *then* $\Sigma; \Gamma, A \vdash t : B$.

Lemma 3.16. *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 3.10) $\Sigma \sqsubseteq \Sigma$.
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 6 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 ABS rule twice). □

In §2.3 a metavariable is instantiated to a term of a mismatched type. The constraints placed on the metavariable instantiation rule are intended to preclude this.

Statement 7 (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 8 (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 generalizes to other judgments; in particular, if* $\Sigma_1, \alpha : A, \Sigma_2$ **sig** *then* $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$.

Statement 9 (Typing of abstractions). *If* $\alpha : A \in \Sigma$, $\Sigma; \Gamma \vdash \alpha\, \vec{x}^n : B$, *where all the variables in the vector* $\vec{x}^n$ *are distinct,* $\Sigma; \Gamma \vdash t : B$, *and* $\textsc{fv}(t) \subseteq \{\vec{x}^n\}$, *then* $\Sigma; \cdot \vdash \lambda^n.(t[\vec{x}^n \mapsto n-1, ..., 0]) : A$ *and* $\lambda^n.(t[\vec{x}^n \mapsto n-1, ..., 0])\ @\ \vec{x}^n = t$.

Statement 10 (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 11 (Context weakening). *If* $\Sigma; \cdot \vdash t : A$ *and* $\Sigma \vdash \Gamma$ **ctx**, *then* $\Sigma; \Gamma \vdash t : A$.

Lemma 3.17. *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 3.12 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 9 we have $\Sigma; \cdot \vdash \lambda^n.t' : A$. By Statement 8 $\textsc{consts}(A) \subseteq \textsc{decls}(\Sigma_1)$. Statement 7 implies that $\Sigma_1; \cdot \vdash \lambda^n.t' : A$, so Σ' **sig**. By Definition 3.10 we get that $\Sigma \sqsubseteq \Sigma'$.

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

(iii) Assume $\Sigma'' \sqsupseteq \Sigma'$.
By Statement 9 we get that $\Sigma; \cdot \vdash \lambda^n.t' : A$ (as mentioned above) and $\lambda^n.t'\ @\ \vec{x} = t$. By Statement 1 $\Sigma' \vdash \Gamma_1$ **ctx**. By Statement 11 and Statement 3 we have $\Sigma'; \Gamma_1 \vdash \lambda^n.t' : A$. By rule $\textsc{meta}_2$ $\Sigma'; \Gamma_1 \vdash \alpha \Rightarrow A$. By Statement 3 and Statement 10 $\Sigma'; \Gamma_1 \vdash t : B_1$. Thus, by rule DELTA-META and Statement 3, $\Sigma'; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1$. By Statement 3 $\Sigma''; \Gamma_1 \vdash \alpha\, \vec{x} \equiv t : B_1$. By reflexivity and Statement 3 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 3.13 we can conclude that $\Sigma'' \not\approx \vec{\mathcal{C}}$. □

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

Definition 3.18 (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 unification 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 $\_ \rightsquigarrow \_$.

The issue described in §2.3 can be avoided by using sound rules.

Lemma 3.19 (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* $\Sigma' \not\approx \vec{\mathcal{C}}$. *If* $\vec{\mathcal{C}}$ *is elaborated from well-formed constraints (as in §2.5), then* Σ' *is a well-typed solution to the original problem as defined in §2.2.*

Proof. By induction on the length of the sequence, using the soundness of each rule. The second statement follows by Definition 3.5, Statement 2 and the typing rules. □

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 3.20 (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 3.21 (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 3.22 (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 $\Theta; \cdot \vdash \mathbb{a} : A$, (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 say that a rule schema is *complete* if it preserves compatible metasubstitutions in the following way:

Definition 3.23 (Completeness of a rule schema)**.** A rule schema is complete if, for each rule $\Sigma; \vec{\mathcal{C}} \rightsquigarrow \Sigma'; \vec{\mathcal{D}}$ in the schema 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 $\equiv$). Here Θ_Σ ("Θ restricted to Σ") is a metasubstitution that contains the same declarations as Θ, in the same order, except that declarations of metavariables that are not declared in Σ are omitted.

We have sketched the soundness of some of the rules from §3.3 (see Lemmas 3.15, 3.16 and 3.17). We state that all of these rules are correct:

Statement 12 (Correctness of rules)**.** *The rules described in §3.3 are sound and complete.*

We now state the main correctness property:

Statement 13 (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 each step in the sequence is a sound and complete rule and Σ' has no uninstantiated metavariable declarations. Then Σ'* **sig**, *$\Sigma' \mid\approx \vec{\mathcal{C}}$, and there exists a unique closed solution to $\Sigma; \vec{\mathcal{C}}$ in the sense that there is (i) a metasubstitution Θ such that $\Theta \vDash \Sigma; \vec{\mathcal{C}}$, and (ii) for every $\tilde{\Theta}$, if $\tilde{\Theta} \vDash \Sigma; \vec{\mathcal{C}}$ then $\Theta \equiv \tilde{\Theta}$.*

For a proof of Statements 12 and 13 under certain assumptions about the type theory, see the first author's licentiate thesis [18].

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 (ignoring the fact that some features used to state the problem are not part of the theory described above). All intermediate signatures and constraints are well-formed.

Example 3.24. As in §2.5, define a problem $\Sigma; \mathcal{C}_1, \mathcal{C}_2$:

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

Step 1. By applying Rule Schema 2 to $\mathcal{C}_2$ we get $\Sigma; \mathcal{C}_1, \mathcal{C}_2 \rightsquigarrow \Sigma; \mathcal{C}_1, \mathcal{C}'_2$, where $\mathcal{C}'_2 \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool}, y : \mathbb{F}\,(\mathit{get}\,(\alpha\, x)) \ddagger \mathbb{F}\,\text{true} \vdash \text{None} \cong^? \alpha\, x : \text{BoolOp} \ddagger \text{BoolOp}$.
Step 2. By applying a symmetric variant of Rule Schema 11 to $\mathcal{C}'_2$ we get $\Sigma; \mathcal{C}_1, \mathcal{C}'_2 \rightsquigarrow \Sigma'; \mathcal{C}_1$, where $\Sigma' \stackrel{\text{def}}{=} \mathbb{F} : \text{Bool} \to \text{Set}, \alpha := \lambda y.\text{None} : \text{Bool} \to \text{BoolOp}$.
Step 3. By applying Rule Schema 3 to $\mathcal{C}_1$ we get $\Sigma'; \mathcal{C}_1 \rightsquigarrow \Sigma'; \mathcal{C}'_1$, where $\mathcal{C}'_1 \stackrel{\text{def}}{=} x : \text{Bool} \ddagger \text{Bool} \vdash \mathbb{F}\,\text{true} \to \text{BoolOp} \cong^? \mathbb{F}\,\text{true} \to \text{BoolOp} : \text{Set} \ddagger \text{Set}$.
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 $\Sigma' \vDash \Sigma; \mathcal{C}_1, \mathcal{C}_2$. By Statement 13 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 [24] 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 unifies types before terms, which prevents it from handling Example 2.2 successfully. We have changed Tog's constraint solver so that it uses twin types and rules based on those discussed in §3.3. We call the resulting implementation Tog$^+$. We do not have room to describe exactly how the constraint solver makes use of the rules, but the source code of Tog$^+$ is available to download [19].

Tog$^+$ differs 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 [3]. 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 and Tog$^+$ admit general recursion and negative data types. We ignore issues related to this.

Unordered Signatures Signatures are ordered (Definition 3.1). 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 [1], although such a check could be implemented.

Tog$^+$ also implements a number of optimizations beyond the ones already in Tog, including the following ones:

(i) We keep track of constraints which it is sufficient to satisfy for both sides of a twin type to become equal, and use this information, for instance, when checking the heterogeneous context equality precondition of Rule Schema 11 (metavariable instantiation).

(ii) We use an "unblocker" mechanism, extending the one implemented in Tog [24]. 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 this enables us to implement Rule Schema 1 efficiently.

4.2 Benchmarks

We have benchmarked the implementation using an example based on McBride's method for representing dependently typed languages inside dependently typed languages [25]. 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, or more complicated than, 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 Tog$^+$ instead uses specific cases such as $\Sigma; \Gamma \vdash \mathrm{Set} \approx \mathrm{Set} : \mathrm{Set}‡\mathrm{Set}$, $\Sigma; \Gamma \vdash \mathrm{Bool} \approx \mathrm{Bool} : \mathrm{Set}‡\mathrm{Set}$, $\Sigma; \Gamma \vdash \mathrm{true} \approx \mathrm{true} : \mathrm{Bool}‡\mathrm{Bool}$ and $\Sigma; \Gamma \vdash \mathrm{false} \approx \mathrm{false} : \mathrm{Bool}‡\mathrm{Bool}$.)

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.

Data and R code used for the figures is available [17].

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 [14]. An algorithm for higher-order unification was proposed by Huet [15]: the algorithm enumerates all possible unifiers but, due to the undecidability of the problem, the algorithm might not terminate. Miller [26] discovered that, when the constraints are of a specific form (the pattern fragment), the problem becomes decidable. Reed [30] presented a terminating algorithm for dynamic pattern unification. Abel and Pientka [5] extended dynamic pattern unification to handle Σ-types with η-equality.

Dependent type checking with metavariables using higher-order unification was implemented in ALF [22, 21]. In ALF intermediate terms are well-typed only modulo a set of constraints. Muñoz [27] shows how to perform unification in dependent type theories such as the Calculus of Constructions [9] in a way that is well-typed at every step. As Norell and Coquand [29, 28] point out, ill-typed terms may cause the type checker to loop. Their solution is to replace possibly ill-typed subterms by guarded constants, terms that do not normalize 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" [16].

Under the OWA, given the signature $\Sigma = \mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}$, whether "$t = \mathbb{a}$ 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 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

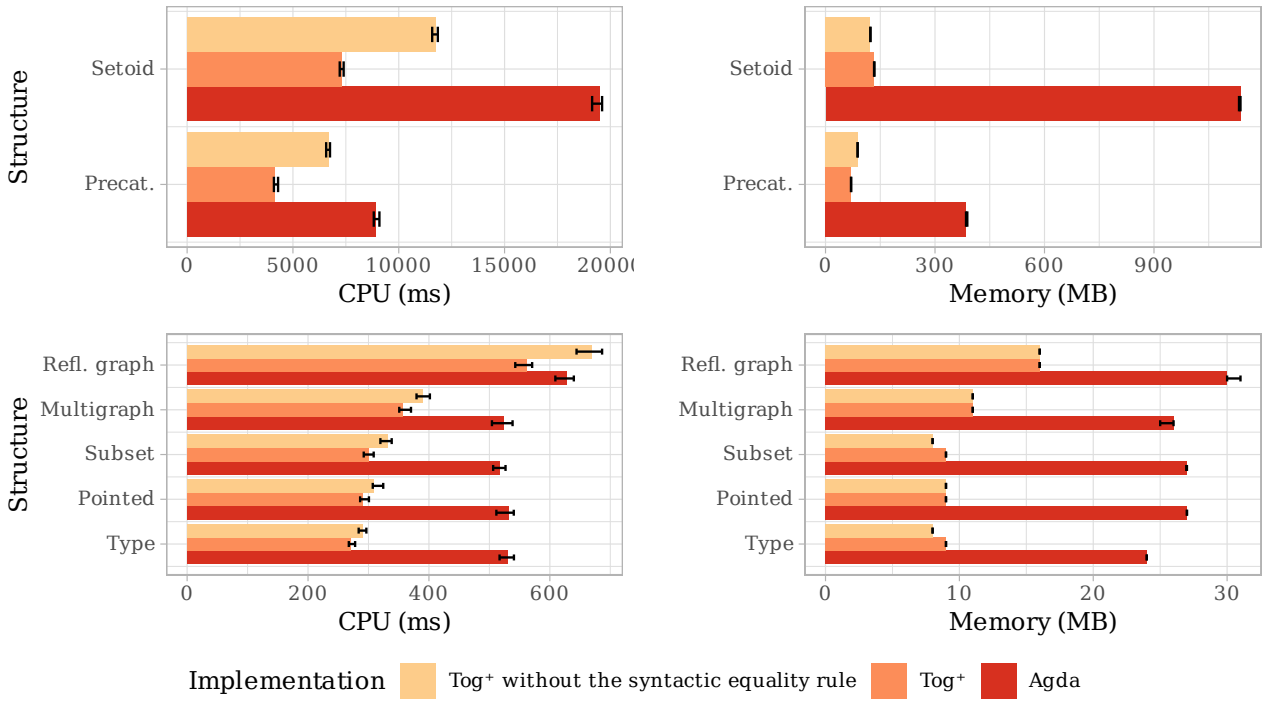

Figure 3. Median resource usage for the examples. The top row shows the resource usage for the two largest examples, while the bottom row shows corresponding information for the rest, using different scales. Each example is benchmarked 40 times. We plot the median value and error bars spanning 95% confidence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

$D = \mathbb{1} : A$, $D = \alpha : A$, 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 3.20) contain no uninstantiated metavariables, similarly to the grounding metasubstitutions defined by Abel and Pientka [5]. We do not believe that the resulting notion of completeness (Definition 3.23) 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 let 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 3) we have that $\Sigma, D; \vec{\mathcal{C}} \rightsquigarrow \Sigma', D; \vec{\mathcal{D}}$ belongs to the same rule schema. (For Rule Schema 10 one can show by induction on the derivation, using Statement 3, 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 [12], is to allow solutions to contain uninstantiated metavariables, and to discuss most general solutions instead of uniqueness.

Agda. Agda uses dynamic pattern unification with postponement [28]: 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 (ignoring bugs). Our prototype uses the same technique. However, in Agda constraints have a single context and a single type. Agda instead uses guarded constants.

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 is convertible to $\mathbb{N}$ if the constraint $\cdot \vdash \mathbb{N} \equiv F\,(\beta\, 0) : \mathrm{Set}$ is solved. 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 [20] 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 [2]. For another open issue [10] we are unaware of any fix not involving a heterogeneous approach to unification such as the one discussed in this paper.

Finally it is not clear how to handle the unit type with η-equality while preserving completeness. Agda is not always complete [3], while we have opted to implement a weaker

notion of pruning and of Rule Schema 9 with the aim of avoiding the issue.

Other Proof Assistants. Other languages/proof assistants based on intensional type theory include Coq [31], Idris [8], Lean [11] and Matita [7]. 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 [13, 12]. 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 include a type-in-type rule. This allows us to support complex examples such as the ones we benchmark (§4.2), which use terms of the form $\Pi\mathsf{Set}B$ and $\Sigma A\mathsf{Set}$ which cannot be typed in their theory. We justify the correctness of our approach under certain assumptions that we hope would hold with a proper stratification of the theory.

Gundry and McBride's judgmental equality is ternary, of the form $\Theta|\Gamma \vdash A \ni t \equiv[v]\equiv u$: "$t$ is equal to u, with η-long standard form v, 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 ($\hat{x} : T_1 \ddagger T_2$) variables.

The heterogeneous equality introduced above (Definition 3.5) 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 some complex examples (§4.2). However, we have not proved that these extensions are implemented correctly.

Gundry and McBride's approach is formulated with fully $\beta\delta$-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.

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 [12, 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. $x : \Pi A_1 ... \Pi A_n \Sigma U_1 U_2 \ddagger \Pi B_1 ... \Pi B_n \Sigma V_1 V_2$).

Finally we can mention that Gundry gives criteria to detect, in some cases, whether a constraint is unsolvable.

6 Conclusions

We have presented an approach to higher-order unification with dependent types. Our approach is based on that of Gundry and McBride [13, 12], but it does not use twin variables. Under certain assumptions—that we hope would hold in a fully stratified version of the theory—the solutions produced are well-typed (see Lemma 3.19), 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).

We have tested the approach by adapting an existing type checker for a tiny variant of Agda. The implementation can handle at least one example that Coq, Idris, Lean and Matita cannot. It can also handle an example that a previous version of Agda struggled with in time and space comparable to or better than that used by the current version of Agda.

Acknowledgments

We want to thank Andreas Abel, Jesper Cockx, Ulf Norell and Andrea Vezzosi for discussions about higher-order unification 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 some anonymous reviewers for useful feedback.

The second author has been supported by a grant from the Swedish Research Council (621-2013-4879).

References

[1] Andreas Abel, Jesper Cockx, and Nils Anders Danielsson. 2015. Agda allows "very dependent" types. Agda Issue #1556. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/1556

[2] Andreas Abel, Jesper Cockx, Nils Anders Danielsson, and Víctor López Juan. 2020. Regression related to fix of #3027. Agda Issue #4408. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/4408

[3] Andreas Abel, Nils Anders Danielsson, and Víctor López Juan. 2017. Overzealous pruning (reprise). Agda Issue #2876. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/2876

[4] Andreas Abel, Martin Stone Davis, Ulf Norell, et al. 2017. (No longer an) Internal error at src/full/Agda/TypeChecking/Substitute.hs:98.

Agda Issue #2709. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/2709

[5] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications (TLCA 2011)*. https://doi.org/10.1007/978-3-642-21691-6_5

[6] Robin Adams. 2004. *A modular hierarchy of logical frameworks*. Ph.D. Dissertation. Faculty of Engineering and Physical Sciences, University of Manchester. Retrieved 2020-06-30 from https://repository.royalholloway.ac.uk/items/2fa04c91-c933-8da6-3bc4-9d300b20cc54/10/

[7] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita interactive theorem prover. In *International Conference on Automated Deduction (CADE 2011)*. https://doi.org/10.1007/978-3-642-22438-6_7

[8] 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

[9] Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95 – 120. https://doi.org/10.1016/0890-5401(88)90005-3

[10] Nils Anders Danielsson and Ulf Norell. 2014. Inconsistent constraints leading to violated invariants in conversion checking. Agda Issue #1467. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/1467

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

[12] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. Department of Computer and Information Sciences, University of Strathclyde. Retrieved 2020-06-30 from http://adam.gundry.co.uk/pub/thesis/

[13] Adam Gundry and Conor McBride. 2012. A tutorial implementation of dynamic pattern unification. Unpublished. Retrieved 2020-06-30 from http://adam.gundry.co.uk/pub/pattern-unify/

[14] 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

[15] 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

[16] C. Maria Keet. 2013. Open World Assumption. In *Encyclopedia of Systems Biology*. 1567–1567. https://doi.org/10.1007/978-1-4419-9863-7_734

[17] Víctor López Juan. 2020. Benchmark data for "Practical Dependent Type Checking Using Twin Types". https://doi.org/10.5281/zenodo.3924267

[18] Víctor López Juan. 2020. *Practical Unification for Dependent Type Checking*. Licentiate Thesis (draft). Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. Retrieved 2020-06-30 from https://lopezjuan.com/project/licentiate/

[19] Víctor López Juan, Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. 2020. Tog$^+$. https://doi.org/10.5281/zenodo.3924250

[20] Víctor López Juan and Ulf Norell. 2018. Internal error in the presence of unsatisfiable constraints. Agda Issue #3027. Retrieved 2020-06-30 from https://github.com/agda/agda/issues/3027

[21] Lena Magnusson. 1994. *The Implementation of ALF – a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. Dissertation. Department of Computer Science, Chalmers University of Technology.

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

[23] Francesco Mazzoli and Andreas Abel. 2016. Type checking through unification. arXiv:1609.09709v1

[24] Francesco Mazzoli, Nils Anders Danielsson, Ulf Norell, Andrea Vezzosi, Andreas Abel, et al. 2017. Tog - A prototypical implementation of dependent types. Retrieved 2020-06-30 from https://github.com/bitonic/tog

[25] Conor McBride. 2010. Outrageous but Meaningful Coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP'10)*. https://doi.org/10.1145/1863495.1863497

[26] 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

[27] 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

[28] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. Retrieved 2020-06-30 from http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[29] Ulf Norell and Catarina Coquand. 2007. Type checking in the presence of meta-variables. Unpublished. Retrieved 2020-06-30 from http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html

[30] 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)*. 49–56. https://doi.org/10.1145/1577824.1577832

[31] The Coq Development Team. 2020. *Coq 8.11.0*. Retrieved 2020-06-30 from https://coq.inria.fr/news/coq-8-11-0-is-out.html

[32] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2003. A concurrent logical framework: The propositional fragment. In *International Workshop on Types for Proofs and Programs (TYPES 2003)*. 355–377. https://doi.org/10.1007/978-3-540-24849-1_23

[33] 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