Type checking in the presence of meta-variables

Ulf Norell
University of Gothenburg

Type checking in the presence of meta-variables

Ulf Norell and Catarina Coquand

Department of Computer Science and Engineering
Chalmers University of Technology
{ulfn,catarina}@cs.chalmers.se

Abstract. In this paper we present a type checking algorithm for a dependently typed logical framework extended with meta-variables. It is common for such frameworks to accept that unification creates substitutions that are not well typed [4, 6, 16], but we give a novel approach to the treatment of meta-variables where well-typedness of substitutions is guaranteed. To ensure type correctness the type checker creates an optimal well-typed approximation of the term being type checked. We use a restricted form of pattern unification, but we believe that the results carry over to other unification algorithms. We prove that the algorithm is sound and terminating. The proposed algorithm has been implemented with promising results.

1 Introduction

Systems based on proposition-as-types provide an elegant approach to interactive proof assistants: the problem of proof checking is reduced to type checking and these systems combine in a natural way deduction and computation. A well understood formulation relies on lambda calculus with dependent types, [14, 1, 3]. The main problem is then checking the judgement $M : A$ expressing that a given term (proof), M, is of type (is a correct proof of the proposition) A.

A type checking algorithm can be naturally divided in two stages[3]. In the first stage we go through the terms and whenever we type check a term M of type A against a type B we collect the equality constraint $A = B$. In the second phase we check these constraints by verifying that the terms are convertible with each other. With dependent types it is important to check the constraints in the right order, and to fail as soon as an equality is invalid, since well typedness of a constraint may dependend on previous constraints being satisfied.

For representing proof search in these frameworks it is convenient to extend the notion of terms with *meta-variables* that stands for yet undetermined terms (proofs). Meta variables are also useful for structure editing, as placeholders for information to be filled in by the user. In this paper we will however focus on type reconstruction where meta-variables are used for representing omitted information that can be recovered from typing constraints through unification.

When adding meta-variables equality checking gets more complicated, since we cannot always decide the validity of an equality, and we may be forced to keep it as a constraint. This is well-known in higher order unification[7]: the constraint $? \ 0 = 0$ has two solutions $? = \lambda x.x$ and $? = \lambda x.0$. This appears also in type theory with constraints of the form $F \ ? = \mathit{Bool}$ where F is defined by computation rules. The fact that we type check modulo yet unsolved constraints can lead to ill-typed terms. For instance, consider the type-checking problem $\lambda g.\ g\ 0 : ((x : F\ ?) \rightarrow F\ (\neg\ x)) \rightarrow \mathit{Nat}$[1] where the term ? is a meta-variable of type Bool, $0{:}\mathit{Nat}$, and $F : \mathit{Bool} \rightarrow \mathsf{Set}$ where $F\ \mathit{false} = \mathit{Nat}$ and $F\ \mathit{true} = \mathit{Bool}$. First we check that $((x : F\ ?) \rightarrow F\ (\neg\ x)) \rightarrow \mathit{Nat}$ is a well-formed type, which generates the constraint $F\ ? = \mathit{Bool}$, since the term $\neg\ x$ forces x to be of type Bool. Checking $\lambda g.\ g\ 0$ against the type $((x{:}F\ ?) \rightarrow F\ (\neg\ x)) \rightarrow \mathit{Nat}$ generate then the constraints $F\ ? = \mathit{Nat}$ and then $F\ (\neg\ 0) = \mathit{Nat}$, which contains an ill-typed term[2].

This problem has some negative consequence for the typechecking algorithm. With dependent types, verifying convertibility between two terms relies on normalising these terms, which is only safe if these terms are well typed. But, as we have seen, in presence of meta-variables, we may not be sure that these terms are well typed, and the typechecker may loop. Furthermore, producing ill-typed terms is not very elegant. It is still the case however that if all constraints can be solved, then we have a correct solution; so we have some form of "partial correctness" and this is indeed the approach taken in [8, 2]. In [6], a similar problem of generating ill-typed terms occur. This is however less problematic in his context, since these terms can still be shown to be normalisable in the logical framework he uses, which is more restricted than the one we consider. Another problem is that when we get a constraint of the form $? = M$, we cannot be sure that M is a solution for ?, since we are not sure that M is well-typed. In [8, 2, 10] this difficulty is avoided by retypechecking M at this point, which is costly.

The main contribution of this paper is to present a type checking algorithm which produces only well-typed constraints for a logical framework extended with meta-variables. The main idea is, for a type-checking

[1] The notation $(x : A) \rightarrow B\ x$ should be read as $\forall x : A.B\ x$.

[2] In fact we will also get the constraint $\mathit{Bool} = \mathit{Bool}$ which is trivially valid and therefore left out.

problem $N : C$, to produce an optimal well-typed approximation N' of N. Whenever we need to verify $M : B$, for a subterm $M : A$ of N, where we cannot yet solve $A = B$, we replace the subterm M by a *guarded constant* p of type B. This constant p will compute to M only when the constraint $A = B$ has been solved. The approximated term N' is in a trivial way well-typed the logical framework without meta-variables. In the example above the type $(x : F\ ?) \to F\ (\mathit{not}\ x)$ will be replaced by $(x : F\ ?) \to F\ (p\ x)$ where $p\ x : \mathit{Bool}$ will compute to $\mathit{not}\ x$ when the meta-variable is replaced with the term *true*.

One interesting application of our work is implicit syntax which allows for a more compact and readable representation of terms. In [11] they show that terms where type information is omitted is more efficient to validate than type checking the complete proof term. This is only possible if constraints are known to be well typed. Their work differs from ours in that they consider a weaker framework where the constraint solving is guaranteed to succeed. The algorithm that we present has been implemented and we have made experiments with examples with several hundreds of meta-variables, which shows that our algorithm scales up to at least medium sized problems.

2 The underlying logic MLF

We use Martin-Löf's logical framework [13] as the underlying logic. The choice of underlying logic is not crucial–the type checking algorithm presented in this paper can be extended to more feature-rich logics with, for instance, recursive definitions, pattern matching, and universe hierachies. See Section 3.3 for a note on how to extend it to pattern matching.

Syntax The syntax of **MLF** is given by the following grammar.

$$\begin{array}{llr}
A, B & ::= \mathsf{Set} \ \mid\ M \ \mid\ (x : A) \to A & \textit{types} \\
M, N & ::= x \ \mid\ c \ \mid\ M\,M \ \mid\ \lambda x.M & \textit{terms} \\
\Gamma, \Delta & ::= () \ \mid\ \Gamma, x : A & \textit{contexts} \\
\Sigma & ::= () \ \mid\ \Sigma, c : A \ \mid\ \Sigma, c : A = M & \textit{signatures}
\end{array}$$

We assume countable sets of variables and constants and we identify terms up to α-conversion. We adopt the convention that variables in contexts are distinct. Similarly a constant may not be declared in a signature more than once.

Repeated application $M\,N_1\ \ldots\ N_k$ is abbreviated $M\,\bar{N}$. Given a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ we sometimes write $\lambda\Gamma.M$ for $\lambda x_1. \ldots \lambda x_n.M$

and $M\,\Gamma$ for $M\,\bar{x}$. Capture avoiding substitution of N for x in M is written $M[N/x]$, or $M[N]$ when x is clear from the context. For dependent function types $(x : A) \to B$, we write $A \to B$ when x is not free in B. The signature contains axioms and non-recursive definitions.

Judgements The type system of **MLF** is presented in six mutually dependent judgement forms.

$\vdash_\Sigma$	Σ is a valid signature
$\Gamma \vdash_\Sigma$	Γ is a valid context
$\Gamma \vdash_\Sigma A$ **type**	A is a valid type in Γ
$\Gamma \vdash_\Sigma M : A$	M has type A in Γ
$\Gamma \vdash_\Sigma A = B$	A and B are convertible types in Γ
$\Gamma \vdash_\Sigma M = N : A$	M and N are convertible terms of type A in Γ

The typing rules follows standard presentations of type theory [13].

3 The type checking algorithm

In this section we present the type checking algorithm for **MLF** with meta-variables.

First we extend the syntax of signatures to include guarded constants and add a new syntactic category for user expressions:

$$\begin{aligned}
C &::= \Gamma \vdash A = B \;\mid\; \Gamma \vdash M = N : A \;\mid\; \Gamma \vdash \bar{M} = \bar{N} : \Delta \\
\Sigma &::= \ldots \;\mid\; \Sigma, p : A = M \ \textbf{when}\ \mathcal{C} \\
e &::= \lambda x.e \;\mid\; x\,\bar{e} \;\mid\; c\,\bar{e} \;\mid\; \mathsf{Set} \;\mid\; (x : e) \to e \;\mid\; ?
\end{aligned}$$

The input to the type checking algorithm is a user expression which could represent either a type or a term. Apart from the usual constructions user expressions can also contain ? representing a meta-variable. During type checking user expressions are translated into **MLF** terms where meta-variables are represented as fresh constants. Note that since we have domain free lambda abstractions we cannot type check β-redexes. Hence the syntax of user expressions disallows them.

A constraint C is an equality constraint that has been postponed because not enough information was available about the meta-variables. Since our conversion checking algorithm is typed the constraints must also be typed. The constraints show up in the signature as guards to guarded constants. We write $p : A = M$ **when** $\mathcal{C}$ for a guarded constant p of type A and value M guarded by the set of constraints $\mathcal{C}$. We have the computation rule that p computes to M when $\mathcal{C}$ is the empty set.

We use the naming convention that lowercase greek letters $\alpha, \beta, \ldots$ stand for constants representing meta-variables and p and q for guarded constants.

3.1 Operations on the signature

All rules work on a signature Σ, containing previously defined constants, meta-variables, and guarded constants. In other words we can write all judgements on the form $\langle\Sigma\rangle\ J \Longrightarrow \langle\Sigma'\rangle$. To make the rules easier to read we first define a set of operations reading and modifying the signature and when presenting the algorithm simply write J for the judgement above. In rules with multiple premisses the signature is threaded top-down, left-to-right.

$$\begin{array}{ll}
\langle\Sigma\rangle\ \mathsf{AddMeta}(\alpha : A) \Longrightarrow \langle\Sigma,\, \alpha : A\rangle & \text{if } \alpha \notin \Sigma \\
\langle\Sigma\rangle\ \alpha := M \Longrightarrow \langle\Sigma_1,\, \alpha : A = M,\, \Sigma_2\rangle & \text{if } \Sigma \ = \ \Sigma_1,\, \alpha : A,\, \Sigma_2 \\
 & \\
\langle\Sigma\rangle\ \mathsf{AddConst}(p : A = M \ \textbf{when}\ \mathcal{C}) \Longrightarrow & \langle\Sigma,\, p : A = M \ \textbf{when}\ \mathcal{C}\rangle \\
 & \text{if } p \notin \Sigma \\
 & \\
\langle\Sigma\rangle\ \mathsf{InScope}_\alpha(M) \Longrightarrow \langle\Sigma\rangle & \text{if } \Sigma \ = \ \Sigma_1,\, \alpha : A,\, \Sigma_2 \ \text{ and} \\
 & \quad c \in M \ \text{ implies } \ c \in \Sigma_1
\end{array}$$

Fig. 1. Operations on the signature

We introduce two operations to manipulate meta-variables: $\mathsf{AddMeta}(\alpha : A)$ adds a new meta-variable α of type A to the signature, and $\alpha := M$ instantiates α to M. For guarded constants we just add the operation $\mathsf{AddConst}(p : A = M \ \textbf{when}\ \mathcal{C})$ to add a new guarded constant to the signature. In Section 3.2 we explain the rules for solving the constraints of a guarded constant. We also introduce an operation $\mathsf{InScope}_\alpha(M)$ to check that M is in scope at the definition site of α (to ensure that α can be instantiated to M). Detailed definitions of the operations can be found in Figure 1.

3.2 The algorithm

Next we present the type checking algorithm. We use a bidirectional algorithm, consisting of the following main judgement forms.

$\Gamma \vdash e$ **type** $\rightsquigarrow A$	well-formed types
$\Gamma \vdash e \uparrow A \rightsquigarrow M$	type checking
$\Gamma \vdash e \downarrow A \rightsquigarrow M$	type inference
$\Gamma \vdash A = B \rightsquigarrow \mathcal{C}$	type conversion
$\Gamma \vdash M = N : A \rightsquigarrow \mathcal{C}$	term conversion

The rules for well-formed types and type checking and inference take a user expression and produce a type or term in **MLF** which is a well-typed approximation of the user expression. Conversion checking produces a set of unsolved constraints which needs to be solved for the judgement to be true in **MLF**.

We use typed conversion for two reasons: it is a nice way to implement η-equality, and perhaps more importantly to prove the correctness of the algorithm we need the invariant that when checking $\Gamma \vdash M = N : A \rightsquigarrow \mathcal{C}$ we have $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, so we need to record the type to make sure the invariant is preserved.

When checking conversion we also need the following judgement forms.

$\Gamma \vdash M \doteq N : A \rightsquigarrow \mathcal{C}$	conversion of weak head normal forms
$\Gamma \vdash \bar{M} = \bar{N} : \Delta \rightsquigarrow \mathcal{C}$	conversion of sequences of terms

Type checking with dependent types involves normalising arbitrary (type correct) terms, so we need to know how to normalise terms in a signature containing meta-variables and guarded constants. We do this by translating the signature to **MLF** and performing the normalisation in **MLF**.

Definition 1. *Given a signature Σ containing meta-variables and guarded constants we define its* **MLF** *restriction $|\Sigma|$ by replacing guarded constants with normal constants, replacing $p : A = M$* **when** *$\mathcal{C}$ by $p : A = M$ if $\mathcal{C}$ is empty, and $p : A$ otherwise.*

The correctness of the type checking algorithm relies on the invariant that when $\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow M \implies \langle\Sigma'\rangle$, we have $\Gamma \vdash_{|\Sigma'|} M : A$ (see Theorem 1).

We write $\langle\Sigma\rangle\ M \rightarrow_{whnf} M' \implies \langle\Sigma\rangle$ if M' is the weak head normal form of M in $|\Sigma|$. Similarly $M \rightarrow_{nf} M'$ means that M' is the normal form of M.

Type checking rules To save some space we omit the rules for checking well-formed types and most of the rules for type checking and inference. The rules are simple extensions of standard type checking algorithms to produce well-typed terms. The interesting type checking rules are the rule for type checking meta-variables and the conversion rules.

$$\frac{\mathsf{AddMeta}(\alpha : \Gamma \to A)}{\Gamma \vdash ? \uparrow A \rightsquigarrow \alpha\,\Gamma} \qquad \frac{\begin{array}{l}\Gamma \vdash e \downarrow B \rightsquigarrow M\\ \Gamma \vdash A = B \rightsquigarrow \emptyset\end{array}}{\Gamma \vdash e \uparrow A \rightsquigarrow M} \qquad \frac{\begin{array}{l}\Gamma \vdash e \downarrow B \rightsquigarrow M\\ \Gamma \vdash A = B \rightsquigarrow \mathcal{C} \neq \emptyset\\ \mathsf{AddConst}(p : \Gamma \to A = \lambda\Gamma.M \ \textbf{when}\ \mathcal{C})\end{array}}{\Gamma \vdash e \uparrow A \rightsquigarrow p\,\Gamma}$$

When type checking a user meta-variable we create a fresh meta-variable, add it to the signature and return it. Since meta-variables are part of the signature they have to be lifted to the top-level.

We have two versions of the conversion rule. The first corresponds to the normal conversion rule and applies when no constraints are generated. The interesting case is when we cannot safely conclude that $A = B$, in which case we introduce a fresh guarded constant. As meta-variables, guarded constants are lifted to the top-level.

Conversion rules When checking conversion of two function types, an interesting question is what to do when comparing the domains gives rise to constraints. The rule in question is

$$\frac{\begin{array}{l}\Gamma \vdash A_1 = A_2 \rightsquigarrow \mathcal{C}, \quad \mathcal{C} \neq \emptyset\\ \mathsf{AddConst}(p : \Gamma \to A_1 \to A_2 = \lambda\Gamma\, x.x \ \textbf{when}\ \mathcal{C})\\ \Gamma, x : A_1 \vdash B_1 = B_2[p\ \Gamma\ x] \rightsquigarrow \mathcal{C}'\end{array}}{\Gamma \vdash (x : A_1) \to B_1 = (x : A_2) \to B_2 \rightsquigarrow \mathcal{C} \cup \mathcal{C}'}$$

To ensure the correctness of the algorithm we need to maintain the invariant that when we check $\vdash A = B \rightsquigarrow \mathcal{C}$ we have $\vdash A$ **type** and $\vdash B$ **type**. Thus if we do not know whether $A_1 = A_2$ it is not correct to check $x : A_1 \vdash B_1 = B_2 \rightsquigarrow \mathcal{C}'$ since B_2 is not well-formed in the context $x : A_1$. To solve the problem we substitute a guarded constant $p\,x$ for x in B_2, where $p\,x$ reduces to x when A_1 and A_2 are convertible.

Term conversion rules Checking conversion of terms is done on weak head normal forms. The only rule that is applied before weak head normalisation is the η-rule.

$$\frac{\Gamma, x : A \vdash M\,x = N\,x : B \rightsquigarrow \mathcal{C}}{\Gamma \vdash M = N : (x : A) \to B \rightsquigarrow \mathcal{C}} \qquad \frac{\begin{array}{l}M \to_{\mathit{whnf}} M'\\ N \to_{\mathit{whnf}} N' \quad \Gamma \vdash M' \doteq N' : A \rightsquigarrow \mathcal{C}\end{array}}{\Gamma \vdash M = N : A \rightsquigarrow \mathcal{C}}$$

In **MLF** function types are not terms so a meta-variable can never be instantiated to a function type. If this was the case we would have to check if the type was a meta-variable, and if so postpone the constraint, since we would not know whether or not the η-rule should be applied.

The weak head normal forms we compare will be of atomic type and so they are of the form $h\,\bar{M}$ where the head h is a variable, constant, meta-variable, or guarded constant. If both terms have the same variable or constant head $h : \Delta \to A$ we compare the arguments in Δ.

$$\frac{h : \Delta \to B \quad \Gamma \vdash \bar{M} = \bar{N} : \Delta \rightsquigarrow \mathcal{C}}{\Gamma \vdash h\ \bar{M} \doteq h\ \bar{N} : A \rightsquigarrow \mathcal{C}}$$

If the heads are different constants or variables conversion checking fails. If one of the heads is a guarded constant we give up and return the problem as a constraint.

$$\frac{}{\Gamma \vdash p\ \bar{M} \doteq N : A \rightsquigarrow \{\Gamma \vdash p\ \bar{M} = N : A\}}$$

If one of the heads is a meta variable we use a restricted form of pattern unification, but we believe that our correctness proof can be extended to more powerful unification algorithms, for example [4, 5, 9, 12, 15]. The crucial step is to prove that meta-variable instantiations are well-typed. In the examples we have studied, using meta-variables for implicit arguments, this simpler form of unification seems to be sufficient. The rule for meta-variable instantiation is

$$\frac{\begin{array}{ll} \bar{x}\ \mathit{distinct} & \\ M \to_{nf} M' & \mathsf{InScope}_{\alpha}(\lambda\bar{x}.M') \\ \mathsf{FV}(M') \subseteq \bar{x} & \alpha := \lambda\bar{x}.M' \end{array}}{\Gamma \vdash \alpha\ \bar{x} \doteq M : A \rightsquigarrow \emptyset}$$

Given the problem $\alpha\,\bar{x} = M$ we would like to instantiate α to $\lambda\bar{x}.M$. This is only correct if $\bar{x}$ are distinct variables, M does not contain any variables other than $\bar{x}$, and any constants refered to by M are in scope at the declaration site of α[3]. Now M might refer to meta-variables introduced after α but which have been instantiated. For this reason we normalise M to M' and try to instantiate α to $\lambda\bar{x}.M'$. A possible optimisation might be to only normalise if M contains out-of-scope constants or variables. If any of the premisses in this rule fail or α is not applied only to variables, we return the constraint as it is.

[3] Note that scope checking subsumes the usual occurs check, since constants are non-recursive.

When checking conversion of argument lists, the interesting case is when comparing the first arguments results in some unsolved constraints.

$$\frac{\Gamma \vdash M = N : A \rightsquigarrow \mathcal{C} \neq \emptyset \quad x \in \mathsf{FV}(\Delta)}{\begin{array}{l}\Gamma \vdash M,\, \bar{M} = N,\, \bar{N} : (x : A)\Delta \rightsquigarrow \\ \quad \{\Gamma \vdash M,\, \bar{M} = N,\, \bar{N} : (x : A)\Delta\}\end{array}} \qquad \frac{\begin{array}{l}\Gamma \vdash M = N : A \rightsquigarrow \mathcal{C}_1 \neq \emptyset \\ \Gamma \vdash \bar{M} = \bar{N} : \Delta \rightsquigarrow \mathcal{C}_2 \qquad x \notin \mathsf{FV}(\Delta)\end{array}}{\Gamma \vdash M,\, \bar{M} = N,\, \bar{N} : (x : A)\Delta \rightsquigarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

If the value of the first argument is used in the types of later arguments ($x \in \mathsf{FV}(\Delta)$) we have to stop and produce a constraint since the types of $\bar{M}$ and $\bar{N}$ differ. If on the other hand the types of later arguments are independent of the value of the first argument, we can proceed and compare them without knowing whether the first arguments are convertible.

Constraint Solving So far, we have not looked at when or how the guards of a constant are simplified or solved. In principle this can be done at any time, for instance as a separate phase after type checking. In practise, however, it might be a better idea to interleave constraint solving and type checking. In Section 5 we prove that this can be done safely. Constraint solving amounts to rechecking the guard of a constant and replacing it by the resulting constraints.

3.3 Adding pattern matching

If we have definitions by pattern matching reduction to weak head normal form might be blocked by an uninstantiated meta variable. For instance $\neg\ \alpha$ cannot be reduced to weak head normal form if $\neg$ is defined by $\neg\ \mathit{true} = \mathit{false}$ and $\neg\ \mathit{false} = \mathit{true}$. Since conversion checking is done on weak head normal forms we generate a constraint when encountering a blocked term.

4 Examples

In this section we look at a few examples that illustrates the workings of the type checker.

A simple example First let us look at a very simple example. Consider the signature $\Sigma = \mathit{Nat}{:}\mathsf{Set}, 0{:}\mathit{Nat}, \mathit{id}{:}(A{:}\mathsf{Set}) \to A \to A = \lambda A\ x.\ x, \alpha{:}\mathsf{Set}$ containing a set Nat with an element 0, a polymorphic identity function id, and a meta-variable α of type Set. Now we want to compute M such that $\vdash \mathit{id}\ ?\ 0 \uparrow \alpha \rightsquigarrow M$. To do this one of the conversion rules have to be applied, so the type checker first infers the type of $\mathit{id}\ ?\ 0$.

$$\dfrac{\vdash \mathit{id} \downarrow (A : \mathsf{Set}) \to A \to A \rightsquigarrow \mathit{id} \qquad \vdash\ ? \uparrow \mathsf{Set} \rightsquigarrow \beta \qquad \dfrac{\vdash 0 \downarrow \mathit{Nat} \rightsquigarrow 0 \quad \beta := \mathit{Nat}}{\vdash 0 \uparrow \beta \rightsquigarrow 0}}{\vdash \mathit{id}\ ?\ 0 \downarrow \beta \rightsquigarrow \mathit{id}\ \beta\ 0}$$

The inferred type β is then compared against the expected type α, resulting in the instantiation $\alpha := \mathit{Nat}$. The final signature is $\mathit{Nat} : \mathsf{Set}$, $0 : \mathit{Nat}$, $\mathit{id} : (A : \mathsf{Set}) \to A \to A = \lambda A\ x.\ x, \alpha : \mathsf{Set} = \mathit{Nat}, \beta : \mathsf{Set} = \mathit{Nat}$ and $M = \mathit{id}\ \beta\ 0$. Note that it is important to look up the values of instantiated meta-variables–it would not be correct to instantiate α to β, since β is not in scope at the point where α is declared.

An example with guarded constants In the previous example all constraints could be solved immediately so no guarded constants had to be introduced. Now let us look at an example with guarded constants. Consider the signature of natural numbers with a case principle:

$$\begin{aligned}&\mathit{Nat} : \mathsf{Set}, 0 : \mathit{Nat}, \mathit{suc} : \mathit{Nat} \to \mathit{Nat},\\ &\mathit{caseNat} : (P : \mathit{Nat} \to \mathsf{Set}) \to P\ 0 \to\\ &\qquad\qquad ((n : \mathit{Nat}) \to P\ (\mathit{suc}\ n)) \to\\ &\qquad\qquad (n : \mathit{Nat}) \to P\ n\end{aligned}$$

In this signature we want to check that $\mathit{caseNat}\ ?\ 0\ (\lambda n.\ n)$ has type $\mathit{Nat} \to \mathit{Nat}$. The first thing that happens is that the arguments to $\mathit{caseNat}$ are checked against their expected types. Checking ? against $\mathit{Nat} \to \mathsf{Set}$ introduces a fresh meta-variable

$$\alpha : \mathit{Nat} \to \mathsf{Set}$$

Next the inferred type of 0 is checked against the expected type $\alpha\ 0$. This produces an unsolved constraint $\alpha\ 0 = \mathit{Nat}$, so a guarded constant is introduced:

$$p : \alpha\ 0 = 0 \ \textbf{when}\ \alpha\ 0 = \mathit{Nat}$$

Similarly, the third argument introduces a guarded constant.

$$q : (n : \mathit{Nat}) \to \alpha\ (\mathit{suc}\ n) = \lambda n.\ n\ \textbf{when}\ (n : \mathit{Nat}) \vdash \alpha\ (\mathit{suc}\ n) = \mathit{Nat}$$

The resulting type correct approximation is $\mathit{caseNat}\ \alpha\ p\ (\lambda n.\ q\ n)$ of type $(n : \mathit{Nat}) \to \alpha\ n$. This type is compared against the expected type $\mathit{Nat} \to \mathit{Nat}$ giving rise to the constraint $\alpha\ n = \mathit{Nat}$ which is solvable with $\alpha = \lambda n.\ \mathit{Nat}$. Once α is instantiated we can perform a $\mathsf{SolveConstraints}$ step to solve the guards on p and q and subsequently reduce $\mathit{caseNat}\ \alpha\ p\ (\lambda n.\ q\ n)$ to $\mathit{caseNat}\ (\lambda n.\ \mathit{Nat})\ 0\ (\lambda n.\ n) : \mathit{Nat} \to \mathit{Nat}$.

What could go wrong? So far we have only looked at type correct examples, where nothing bad would have happened if we had not introduced guarded constants when we did. The following example shows how things can go wrong. Take the signature $\mathit{Nat} : \mathsf{Set}, 0 : \mathit{Nat}$. Now add the perfectly well-typed identity function *coerce*:

$\mathit{coerce} : (F : \mathit{Nat} \to \mathsf{Set}) \to F\ 0 \to F\ 0 = \lambda F\ x.\ x$

For any well-typed term $t : B$ and type A, $\mathit{coerce}\ ?\ t$ will successfully check against A, resulting in the constraints $\alpha\ 0 = B$ and $A = \alpha\ 0$, none of which can be solved. If we did not introduce guarded constants $\mathit{coerce}\ ?\ t$ would reduce to t and hence we could use *coerce* to give an arbitrary type to a term. For instance we can type[4]

$$\begin{aligned} \omega &: (N \to N) \to N = \lambda x.\ x\ (\mathit{coerce}\ ?\ x) \\ \Omega &: N \qquad\qquad\quad\ = \omega\ (\mathit{coerce}\ ?\ \omega) \end{aligned}$$

where without guarded constants Ω would reduce to the non-normalising λ-term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. With our, algorithm new guarded constants are introduced for for the argument to *coerce* and for the application of coerce. So the type correct appoximation of Ω would be $\omega\ p$ where $p = \mathit{coerce}\ \alpha\ q$ **when** $\alpha\ 0 = N \to N$ and $q = \omega$ **when** $(N \to N) \to N = \alpha\ 0$.

5 Proof of correctness

The correctness of the algorithm relies on the fact that we only compute with well-typed terms. This guarantees the existence of normal forms, and hence, ensures the termination of the type checking algorithm.

The proof will be done in two stages: first we prove soundness in the absence of constraint solving, and then we prove that constraint solving is sound.

5.1 Soundness without constraint solving

There are a number of things we need to prove: that type checking preserves well-formed signatures, that it produces well-typed terms, that conversion checking is sound, and that new signatures respect the old signatures. Unfortunately these properties are all interdependent, so we cannot prove them separately.

Definition 2 (Signature extension). *We say that Σ'* extends *Σ if for any* **MLF** *judgement J, $\vdash_\Sigma J$ implies $\vdash_{\Sigma'} J$.*

[4] This only type checks if we allow meta-variables to be instantiated to function types, which is not the case in **MLF**. However, the type checking algorithm can be extended to handle this, something we have done in the implementation.

Note that this definition admits both simple extensions–adding a new constant–and refinement, where we give a definition to a constant.

Now we are ready to state the soundness of the type checking algorithm in the absence of constraint solving.

Theorem 1 (Soundness of type checking). *Type checking produces well-typed terms, conversion checking produces well-formed constraints and if no constraints are produced, the conversion is valid in* **MLF***. Also, all rules produce well-formed extensions of the signature.*

Proof. By induction on the derivation. The most interesting case is the meta-variable instantiation case where we have to prove that the instantiation constructs a valid extension of the signature. This is proven by showing that the instantiation is well-typed.

Since well-typed terms in **MLF** have normal forms we get the existence of normal forms for type checked terms and hence the type checking algorithm is terminating.

Corollary 1. *The type checking algorithm is terminating.*

Note that type checking terminates with one of three answers: *yes it is type correct*, *no it is not correct*, or *it might be correct if the meta-variables are instantiated properly.* The algorithm is not complete, since finding correct instantiations to the meta-variables is undecidable in the general case.

5.2 Soundness of constraint solving

In the previous section we proved type checking sound and decidable in the absence of constraint solving. We also mostly ignored the constraints, only requiring them to be well-formed. In this section we prove that the terms produced by the type checker stay well-typed under constraint solving. This is done by showing that constraint solving is a signature extension operator in the sense of Definition 2.

Previously we only ensured that the **MLF** restriction of the signature was well-formed. Now, since we are going to update and remove the constraints of guarded constants we have to strengthen the requirements and demand *consistent* signatures. A signature is consistent if the solution of a guard is a sufficient condition for the well-typedness of the definition it is guarding.

In order to prove that type checking preserves consistency, we first need to know that the constraints we produce are sound.

Lemma 1 (Soundness of generated constraints). *The constraints generated during conversion checking ensures that the checked terms are convertible. For instance, if $\Gamma \vdash A = B \rightsquigarrow \mathcal{C}$, then solving $\mathcal{C}$ guarantees that $\Gamma \vdash A = B$ in* **MLF***.*

Lemma 2 (Type checking preserves consistency). *Type checking and conversion checking preserves consistent signatures.*

Proof. We have to prove that when we introduce a guarded constant the guard ensures the well-typedness of the value. This follows from Lemma 1.

Lemma 3 (Constraint solving is sound). *If Σ is consistent and the solving the constraints yields a signature Σ', then Σ' is consistent and Σ' extends Σ.*

Proof. Follows from Theorem 1, Lemma 1, and Lemma 2.

From this follows that we can mix type checking and constraint solving freely, so we can add a constraint solving rule to the type checking algorithm. In order to obtain optimal approximations we have to solve constraints eagerly, i.e as soon as a meta-variable has been instantiated.

5.3 Relating user expressions and checked terms

An important property of the type checking algorithm is that the type correct terms produced correspond to the expressions being type checked. The correspondance is expressed by stating that the only operations the type checker is allowed when constructing a term is replacing a ? by a term (*refinement*) and replacing a term by a guarded constant (*approximation*).

Lemma 4. *If $\Gamma \vdash e \uparrow A \rightsquigarrow M$ then M approximates a refinement of e. This property is preserved when unfolding instantiated meta-variables and guarded constants in M.*

Lemma 5. *If $\Gamma \vdash e \uparrow A \rightsquigarrow M$ then M is an* optimal approximation *of a refinement M' of e.*

Proof. The proof relies on the fact that we only introduce guarded constants when absolutely necessary and solve the constraints eagerly. This is proven by showing that the constraints produced by conversion checking are not only sufficient but also necessary for the validity of the judgement.

5.4 Main result

We now prove the main soundness theorem stating that if all meta-variables are instantiated and all guards solved, then the term produced by the type checker (extended with constraint solving) is valid in the original signature after unfolding the definitions of the meta-variables and guarded constants introduced during type checking.

Theorem 2 (Soundness of type checking). *If Σ is a well-formed* **MLF** *signature and $\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow M \implies \langle\Sigma'\rangle$, then if all meta-variables have been instantiated and all guards are empty in Σ', then $\Gamma \vdash_\Sigma M\sigma : A$ where σ is the substitution inlining the meta variables and constants in Σ'. Moreover, $M\sigma$ is a refinement of e.*

6 Conclusions and future work

In this paper we have shown how to do type checking for a dependently typed logic extended with meta-variables. To maintain the important invariant that terms being evaluated are type correct we work with well-typed approximations of terms, where potentially ill-typed subterms have been replaced by constants. We showed that type checking is decidable, that the algorithm is sound and that the approximated terms are optimal.

We present the type checking algorithm for a simple dependently typed logical framework **MLF**, but it can be extended to more advanced logics. This is evidenced by the fact that we have implemented the algorithm for the Agda language, supporting for instance, definitions by pattern matching, a hierarchy of universes and constants with variable arity. The algorithm has proven to work well with examples of several hundred meta-variables.

There are two main directions of future work. First extending the correctness proof to a more feature-rich logic. Much of this work has already been done in the implementation but some work remains in working out the details of the proofs. The other direction of future work is to build on top of this algorithm. For instance, a system for implicit arguments or Alf-style interaction[8].

Acknowledgement We would like to thank Conor McBride who generously shared with us how meta-variables are treated in Epigram. In particular we want to thank him for the idea of naming possibly ill-typed terms which simplifies equality reasoning. The authors would also like to thank Thierry Coquand for many valuable comments on this work.

References

1. H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.
2. C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, Sep 1999.
3. N. G. de Bruijn. A plea for weaker frameworks. pages 40–67, 1991.
4. G. Dowek. Higher-order unification and matching. pages 1009–1062, 2001.
5. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings of the Tenth Annual IEEE Symp. on Logic in Computer Science, LICS 1995*, pages 366–374. IEEE Computer Society Press, June 1995.
6. C. M. Elliot. Higher-order unification with dependent function types. In N. Derikowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, pages 121–136, April 1989.
7. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.
8. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.
9. D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Logic Programming: Proc. of the Eighth International Conference*, pages 255–269. MIT Press, Cambridge, MA, 1991.
10. C. Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theor. Comput. Sci.*, 266(1-2):407–440, 2001.
11. G. Necula and P. Lee. Efficient representation and validation of proofs. In *LICS'98*, pages 93–104. IEEE, June 1998.
12. T. Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.
13. B. Nordström, K. Petersson, and J. Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science*, volume 5. OUP, Oct. 2000.
14. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
15. F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, 1991.
16. D. Pym. *Proof, search and computation in general logic.* PhD thesis, Univesity of Edinburgh, 1990.