Towards Unification for Dependent Types

research paper, extended abstract

Ningning Xie and Bruno C. d. S. Oliveira

The University of Hong Kong

Abstract. Unification and subtyping for dependent types are always complicated and even unpredictable. In this paper, we show how to do unification and subtyping for first-order dependent types based on a new strategy called *type sanitization* which helps resolve dependency between types. Our algorithm is remarkably simple and predictable.

1 Introduction

Dependent types are currently increasingly adopted in many language designs due to its expressiveness [13, 8, 11, 9, 10, 2]. However, type inference or unification on those language is not easy. This is because more power a type system has, more sophisticated the type system becomes. The dependency between expressions and types bring lots of complexities.

Existing literature [14, 1, 5] that tries to give specification for type inference or unification of a dependent language is quite complicated, and even becomes non-intuitive or unpredictable once it involves so many constructs or features.

In this paper, we presents an easy strategy to do unification based on alpha-equality for first-order dependent types. This algorithm is based on alpha-equality for following reasons. Firstly, for unification algorithms based on beta-equality, if the system has strong normalization, the algorithm usually reduces all types into normal forms and then compares the normal forms using alpha-equality. Secondly, there are proposals for type system without strong normalization, for example, doing type-level computations using casts [3, 7, 12] . For those systems, the unification is naturally based on alpha-equality.

Our notations to do formalization are inspired by [4]. We come up with a new process called *type sanitization* that helps resolve the dependency problem. Later on, the type sanitization process is extended to deal with restricted polymorphic types. Based on type sanitization, our algorithm are remarkably simple and well-behaved. Though there are no formal proofs for the meta-theory of the system yet (which is still in progress), we give many conjectures that we believe are intuitive.

We expect our algorithm serves as a footstone towards a simple and predictable unification/subtyping algorithm for dependent types. This is also for filling the gap between delicate unification algorithms for simple types and sophisticated unification algorithms for dependent types. A non-goal of our work is to replace existing matured unification/subtyping algorithm. More precisely, our main contributions are:

$\boxed{\Gamma \ ctx}$

$$\frac{}{\varnothing \ ctx}\ \textsc{WC-Empty} \qquad \frac{\Gamma \ ctx \qquad x \notin dom(\Gamma) \qquad \Gamma \vdash \sigma}{\Gamma, x : \sigma \ ctx}\ \textsc{WC-Var}$$

$$\frac{\Gamma \ ctx \qquad \widehat{\alpha} \notin dom(\Gamma)}{\Gamma, \widehat{\alpha} \ ctx}\ \textsc{WC-EVar} \qquad \frac{\Gamma \ ctx \qquad \widehat{\alpha} \notin dom(\Gamma) \qquad \Gamma \vdash \tau}{\Gamma, \widehat{\alpha} = \tau \ ctx}\ \textsc{WC-S-EVar}$$

$\boxed{\Gamma \vdash \sigma}$

$$\frac{\Gamma \vdash \sigma : \star \dashv \Gamma, \Delta}{\Gamma \vdash \sigma}\ \textsc{WF-Other}$$

$\boxed{\Gamma \vDash \sigma}$

$$\frac{x \in \Gamma}{\Gamma \vDash x}\ \textsc{WT-Var} \qquad \frac{\widehat{\alpha} \in \Gamma}{\Gamma \vDash \widehat{\alpha}}\ \textsc{WT-EVar} \qquad \frac{}{\Gamma \vDash \star}\ \textsc{WT-Star}$$

$$\frac{\Gamma \vDash e_1 \qquad \Gamma \vDash e_2}{\Gamma \vDash e_1 \ e_2}\ \textsc{WT-App} \qquad \frac{\Gamma, x : \sigma \vDash e}{\Gamma \vDash \lambda x : \sigma.\ e}\ \textsc{WT-Lam} \qquad \frac{\Gamma \vDash \sigma_1 \qquad \Gamma, x \vDash \sigma_2}{\Gamma \vDash \Pi x : \sigma_1.\ \sigma_2}\ \textsc{WT-Pi}$$

Fig. 1. Well formedness

- We come up with a strategy called *type sanitization* that resolves the dependency between types.
- Based on type sanitization, we give a specification of an alpha-equality based unification algorithm for first-order dependent types.
- We show how to extend type sanitization for a specification of a subtyping algorithm in a language including restricted polymorphic types.

In Section 2, we present a overview of a first-order dependently typed language. In Section 3, we formalize the unification problem and present the type sanitization and unification. In Section 4, we extend the language with polymorphic types, and then present the process of extended type sanitization along with subtyping rules. Finally Section 5 concludes the paper.

2 Language Overview

Below shows the syntax of the system.

Type	σ, τ	$::= \widehat{\alpha} \mid e$
Expr	e	$::= x \mid \star \mid e_1\ e_2 \mid \lambda x : \sigma.\ e \mid \Pi x : \sigma_1.\ \sigma_2$
		$\mid \ \lambda x.\ e \equiv \lambda x : \widehat{\alpha}.\ e$
Contexts	Γ, Θ, Δ	$::= \varnothing \mid \Gamma, x : \sigma \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau$

Expression. Expressions e are variables x, a single sort $\star$ to represent the type of types (with an impredicative axiom $\star : \star$), application $e_1\ e_2$, function $\lambda x : \sigma.\ e$, and Pi type $\Pi x : \tau_1.\ \tau_2$. The unannotated function $\lambda x.\ e$ is equivalent to the same function with the bound variable annotated by a fresh unification variable $\widehat{\alpha}$.

$$
\begin{aligned}
[\varnothing]e &= e \\
[\Gamma, x : \tau]e &= [\Gamma]e \\
[\Gamma, \widehat{\alpha}]e &= [\Gamma]e \\
[\Gamma, \widehat{\alpha} = \tau]e &= [\Gamma](e[\widehat{\alpha} \mapsto \tau])
\end{aligned}
$$

Fig. 2. Applying a context.

Types. Types σ, τ contain all expression forms, with an additional $\widehat{\alpha}$ to denote unification variables.

Contexts. A context is an ordered list of variables and unification variables, which can either be unsolved ($\widehat{\alpha}$) or solved by a type τ ($\widehat{\alpha} = \tau$).

It is important for a context to be ordered to solve the dependency between variables. For example, the expression

$\lambda z.\ \lambda x.\ \lambda y{:}x.\ y == z$

cannot type check because z cannot be of type x since x appears after z.

Figure 1 gives the definition of well formedness of a context ($\Gamma\ ctx$), and the well formedness of a type under certain context ($\Gamma \vdash \sigma$). Both definitions rely on the typing process, which due to the limitation of space is put in the appendix. For a type to be well formed, the typing should not solve existing unification variables. The last judgment, the well-scopedness $\Gamma \vDash \sigma$ means under context Γ, σ is a syntactically legal term with all variables bounded.

Hole notation. We use hole notations like $\Gamma[x]$ to denote that the variable x appears in the context, sometimes it is also written as Γ_1, x, Γ_2.

Multiple holes also keep the order. For example, $\Gamma[x][\widehat{\alpha}]$ not only require the existance of both variable x and $\widehat{\alpha}$, but also require that x appears before $\widehat{\alpha}$.

Hole notation is also used for replacement and modification. For example, $\Gamma[\widehat{\alpha} = \star]$ means the context keeps unchanged except $\widehat{\alpha}$ now is solved by $\star$.

Applying Context. Since the context records all the solutions of solved unification variables, it can be used as a substitution. Figure 2 defines the substitution process, where all solved unification variables are substituted by their solution.

3 Unification

The unification problem is formalized as:

$\Gamma \vdash \tau_1 \simeq \tau_2 \dashv \Theta$

The input of the unification is the current context Γ, and two types τ_1 and τ_2 that are being unified. The output of the unification is a new context Θ which extends the original context with probably more new unification variables or more existing unification variables solved. For example,

$\widehat{\alpha} \vdash \widehat{\alpha} \simeq \mathit{Int} \dashv \widehat{\alpha} = \mathit{Int}$

For a valid unification problem, it must have the invariant: $[\Gamma]\tau_1 = \tau_1$, and $[\Gamma]\tau_2 = \tau_2$. Namely, the input types must be fully applied under the input context. So the following is not a valid unification problem input:

$\widehat{\alpha} = \mathit{String} \vdash \widehat{\alpha} \simeq \mathit{Int}$

We assume this invariant is maintained through the whole formalization.

3.1 Type Sanitization

As we mentioned before, our unification is based on alpha-equality. So in most cases, the unification rules are intuitively structural. The most difficult one which is also the most essential one, is how to unify a unification variable with another type. We discuss those cases first.

Variable Orders matter. While unifying existential variable $\widehat{\alpha}$ and type τ, we tends to directly derive that $\widehat{\alpha} = \tau$. But τ may not be a valid type as the solution for $\widehat{\alpha}$. Consider a unification example:

$\Gamma, \widehat{\alpha}, x \vdash \widehat{\alpha} \simeq x$

which has no feasible solution. Because x is not in the scope of $\widehat{\alpha}$.

From this observation, it seems whenever we have the unification problem as

$\Gamma, \widehat{\alpha}, \Delta \vdash \widehat{\alpha} \simeq \tau$

we need to check if τ is well-scoped in Γ. Only if it satisfies the scope constraint, we can derive $\widehat{\alpha} = \tau$.

Unification variable orders do not matter. However, there are unification problems that even if the type does not satisfies the scope constraint, it still has feasible solutions. Consider:

$\Gamma, \widehat{\alpha}, \widehat{\beta} \vdash \widehat{\alpha} \simeq \Pi x : \widehat{\beta}.\ x$

Here because $\widehat{\beta}$ appears after $\widehat{\alpha}$, we cannot directly derive $\widehat{\alpha} = \Pi x : \widehat{\beta}.\ x$ which is ill typed. But this unification does have a solution context if we could sanitize the appearance of $\widehat{\beta}$ by solving it by a fresh unification variable $\widehat{\alpha}_1$ which is put in the scope of $\widehat{\alpha}$, and then get an equivalent unification problem:

$\Gamma, \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1 \vdash \widehat{\alpha} \simeq \Pi x : \widehat{\alpha}_1.\ x$

Then we can derive the solution of this equivalent unification problem:

$\Gamma, \widehat{\alpha}_1, \widehat{\alpha} = \Pi x : \widehat{\alpha}_1.\ x, \widehat{\beta} = \widehat{\alpha}_1.$

From this example, we can see that the orders of unification variables do not matter because we can always solve it by a fresh unification variable that satisfies the scope constraint. So for a unification problem

$\Gamma, \widehat{\alpha}, \Delta \vdash \widehat{\alpha} \simeq \tau$

$$\boxed{\Gamma[\widehat{\alpha}] \vdash \tau_1 \longmapsto \tau_2 \dashv \Theta}$$

$$\frac{}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash \widehat{\beta} \longmapsto \widehat{\alpha}_1 \dashv \Gamma[\widehat{\alpha}_1, \widehat{\alpha}][\widehat{\beta} = \widehat{\alpha}_1]}\ \text{I-EVar} \qquad \frac{}{\Gamma[\widehat{\beta}][\widehat{\alpha}] \vdash \widehat{\beta} \longmapsto \widehat{\beta} \dashv \Gamma[\widehat{\beta}][\widehat{\alpha}]}\ \text{I-EVar2}$$

$$\frac{}{\Gamma \vdash \star \longmapsto \star \dashv \Gamma}\ \text{I-Star} \qquad \frac{\Gamma \vdash e_1 \longmapsto e_1' \dashv \Theta_1 \qquad \Theta_1 \vdash e_2 \longmapsto e_2' \dashv \Theta}{\Gamma \vdash e_1\ e_2 \longmapsto e_1'\ e_2' \dashv \Theta}\ \text{I-App}$$

$$\frac{}{\Gamma \vdash x \longmapsto x \dashv \Gamma}\ \text{I-Var} \qquad \frac{\Gamma \vdash \sigma \longmapsto \sigma_1' \dashv \Theta_1 \qquad \Theta_1, x : \sigma \vdash e \longmapsto e' \dashv \Theta, x : \sigma}{\Gamma \vdash \lambda x : \sigma.\ e \longmapsto \lambda x : \sigma'.\ e' \dashv \Theta}\ \text{I-Lam}$$

$$\frac{\Gamma \vdash \sigma_1 \longmapsto \sigma_1' \dashv \Theta_1 \qquad \Theta_1, x : \sigma_1 \vdash \sigma_2 \longmapsto \sigma_2' \dashv \Theta, x : \sigma_1}{\Gamma \vdash \Pi x : \sigma_1.\ \ \sigma_2 \longmapsto \Pi x : \sigma_1'.\ \sigma_2' \dashv \Theta_2}\ \text{I-Pi}$$

Fig. 3. Type Sanitization

we need to sanitize the unification variables in τ before we check the scope constraint. We call this process *type sanitization*, which is given in Figure 3. The interesting cases are I-EVar and I-EVar2. In I-EVar, because $\widehat{\beta}$ appears after $\widehat{\alpha}$, so we create a fresh unification variable $\widehat{\alpha}_1$, which is put before $\widehat{\alpha}$, and solve $\widehat{\beta}$ by $\widehat{\alpha}_1$. In I-EVar2, because $\widehat{\beta}$ is in the scope of $\widehat{\alpha}$, so we leave it unchanged.

3.2 Unification

Based on type sanitization, Figure 4 gives the unification rules.

Rule U-AEq corresponds to the case when two types are already alpha-equivalent. Most of the rest rules are structural. Two most subtle ones are rule U-EVarTy and U-TyEVar, which corresponding respectively to when the unification variable is on the left and on the right. We go through the first one. There are three preconditions. First is the occurs check, which is to make sure $\widehat{\alpha}$ does not appear in the free variables of τ_1. Then we use type sanitization to make sure all the unification types in τ_1 that are out of scope of $\widehat{\alpha}$ are turned into fresh ones that are in the scope of $\widehat{\alpha}$. This process gives us the output type τ_2, and output context $\Theta_1, \widehat{\alpha}, \Theta_2$. Finally, τ_2 could also contain variables of which the orders matter, so we use $\Theta_1 \vDash \tau_2$ to make sure τ_2 is well scoped.

Example. Below shows the process for the unification problem $\Gamma, \widehat{\alpha}, \widehat{\beta} \vdash \widehat{\alpha} \simeq \Pi x : \widehat{\beta}.\ x$. For clarity, we denote $\Theta = \Gamma, \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1$. And it is easy to verify $\widehat{\alpha} \notin FV(\Pi x : \widehat{\beta}.\ x)$.

$$\frac{\dfrac{\dfrac{}{\Gamma, \widehat{\alpha}, \widehat{\beta} \vdash \widehat{\beta} \longmapsto \widehat{\alpha}_1 \dashv \Theta}\ \text{I-Evar2} \qquad \dfrac{}{\Theta \vdash x \longmapsto x \dashv \Theta}\ \text{I-Var}}{\Gamma, \widehat{\alpha}, \widehat{\beta} \vdash \Pi x : \widehat{\beta}.\ x \longmapsto \Pi x : \widehat{\alpha}_1.\ x \dashv \Theta}\ \text{I-Pi} \qquad \dfrac{}{\Gamma, \widehat{\alpha}_1 \vDash \Pi x : \widehat{\alpha}_1.\ x}}{\Gamma, \widehat{\alpha}, \widehat{\beta} \vdash \widehat{\alpha} \simeq \Pi x : \widehat{\beta}.\ x \dashv \Gamma, \widehat{\alpha}_1, \widehat{\alpha} = \Pi x : \widehat{\alpha}_1.\ x, \widehat{\beta} = \widehat{\alpha}_1}\ \text{U-EvarTy}$$

$$\boxed{\Gamma \vdash \tau_1 \simeq \tau_2 \dashv \Theta}$$

$$\frac{}{\Gamma \vdash \tau \simeq \tau \dashv \Gamma}\ \text{U-AEq} \qquad \frac{\widehat{\alpha} \notin \mathsf{FV}(\tau_1) \qquad \Gamma[\widehat{\alpha}] \vdash \tau_1 \longmapsto \tau_2 \dashv \Theta_1, \widehat{\alpha}, \Theta_2 \qquad \Theta_1 \vDash \tau_2}{\Gamma[\widehat{\alpha}] \vdash \widehat{\alpha} \simeq \tau_1 \dashv \Theta_1, \widehat{\alpha} = \tau_2, \Theta_2}\ \text{U-EvarTy}$$

$$\frac{\widehat{\alpha} \notin \mathsf{FV}(\tau_1) \qquad \Gamma[\widehat{\alpha}] \vdash \tau_1 \longmapsto \tau_2 \dashv \Theta_1, \widehat{\alpha}, \Theta_2 \qquad \Theta_1 \vDash \tau_2}{\Gamma[\widehat{\alpha}] \vdash \tau_1 \simeq \widehat{\alpha} \dashv \Theta_1, \widehat{\alpha} = \tau_2, \Theta_2}\ \text{U-TyEVar}$$

$$\frac{\Gamma \vdash \tau_2 \simeq \tau_2' \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]\tau_1 \simeq [\Theta_1]\tau_1' \dashv \Theta}{\Gamma \vdash \tau_1\ \tau_2 \simeq \tau_1'\ \tau_2' \dashv \Theta}\ \text{U-App}$$

$$\frac{\Gamma \vdash \tau_1 \simeq \tau_3 \dashv \Theta_1 \qquad \Theta_1, x : \tau_1 \vdash [\Theta_1]\tau_2 \simeq [\Theta_1]\tau_4 \dashv \Theta, x : \tau_1}{\Gamma \vdash \lambda x : \tau_1.\ \tau_2 \simeq \lambda x : \tau_3.\ \tau_4 \dashv \Theta}\ \text{U-LamAnn}$$

$$\frac{\Gamma \vdash \tau_1' \simeq \tau_1 \dashv \Theta_1 \qquad \Theta_1, x : \tau_1 \vdash [\Theta_1]\tau_2 \simeq [\Theta_1]\tau_2' \dashv \Theta, x : \tau_1}{\Gamma \vdash \Pi x : \tau_1.\ \tau_2 \simeq \Pi x : \tau_1'.\ \tau_2' \dashv \Theta}\ \text{U-Pi}$$

Fig. 4. Unification rules

Comparison. In [4], they have no type sanitization because if one side is a unification variable $\widehat{\alpha}$, and the other side is some type constructors, they will decompose the unification problem into several sub-unifications according to the type constructor, until type being unified is in the scope of $\widehat{\alpha}$. If both sides are unification variables, they will take two cases into consideration: the left one appears first in the context, or the right one. For example, a decomposing rule for unification where the left hand side is a function type is:

$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \simeq \tau_1 \dashv \Theta \qquad \Theta \vdash [\Theta]\tau_2 \simeq \widehat{\alpha}_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash \tau_1 \to \tau_2 \simeq \widehat{\alpha} \dashv \Delta}$$

Here the original unification variable $\widehat{\alpha}$ is solved by a function type consisting of two fresh unification variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$, which are then unified with τ_1 and τ_2 sequentially. However, this cannot hold with dependent types. For example consider we apply this rule to this specific unification:

$\Gamma \vdash \widehat{\alpha} \simeq \Pi x : \star.\ x$

It is obvious that $\widehat{\alpha}_2$ should be solved by x. So in order to make it well typed, we need to put x before $\widehat{\alpha}_2$ into the context. However, this means x will remain in the context, and it is available for any later unification variable that should not have access to x. Type sanitization solves this problem by sanitizing Pi type instead of decomposing it.

3.3 Meta-theory

Conjecture 1. if $\Gamma[\widehat{\alpha}] \vdash \sigma_1 \longmapsto \sigma_2 \dashv \Theta$, then $[\Theta]\sigma_1 = [\Theta]\sigma_2$.

Proof Sketch. By induction on type sanitization rules.

Conjecture 2. if $\Gamma \vdash \sigma_1 \simeq \sigma_2 \dashv \Theta$, then $[\Theta]\sigma_1 = [\Theta]\sigma_2$.

Proof Sketch. By induction on unification rules.

Conjecture 3. if $\Gamma[\widehat{\alpha}] \vdash \sigma_1 \longmapsto \sigma_2 \dashv \Theta$, and $\Gamma[\widehat{\alpha}] \vdash \sigma_1$, then $\Theta \vdash \sigma_2$.

Proof Sketch. Type sanitization only replaces some unification variables with fresh ones. And all fresh unification variables are in the scope of the output context. So the output type preserves the well formedness of the input type.

Conjecture 4. if $\Gamma \vdash \sigma_1 \simeq \sigma_2 \dashv \Theta$, $\Gamma \vdash \sigma_1$, and $\Gamma \vdash \sigma_2$, then $\Theta \vdash \sigma_1$, and $\Theta \vdash \sigma_2$.

Proof Sketch. Unification extends the input context with more fresh unification variables and newly solved existing unification variables. Those updates will not change the well formedness of the type. The key point here is to prove rule U-EVarTy and E-TyEVar give well-formed output contexts. This is based on Conjecture 3, from where we know $\Theta_1, \widehat{\alpha}, \Theta_2 \vdash \tau_2$. From another precondition $\Theta_1 \vDash \tau_2$, we know $\widehat{\alpha}, \Theta_2$ plays no role for the well formedness of τ_2. So $\Theta_1 \vdash \tau_2$.

4 Extended with Polymorphism

In this section, we discuss how to extends the type sanitization to deal with polymorphic types. Our system has two simplifications: polymorphic type variables are of type $\star$, and subtyping only concerns Pi types and polymorphic types.

4.1 Language

The new definition of syntax extends the original one:

Type	σ	$::= \widehat{\alpha} \mid e$
Expr	e	$::= x \mid \star \mid e_1\ e_2 \mid \lambda x : \sigma.\ e \mid \Pi x : \sigma_1.\ \sigma_2$
		$\mid\ \lambda x.\ e \equiv \lambda x : \widehat{\alpha}.\ e$
		$\mid\ \forall x : \star . \sigma$
Monotype	τ	$::= \{\sigma' \in \sigma, \forall \notin \sigma'\}$
Contexts	Γ, Θ, Δ	$::= \varnothing \mid \Gamma, x : \sigma \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau$

Expression. Expressions e are extended with the polymorphic type $\forall x : \star . \sigma$. Note our definition of polymorphic type is not the general version $\forall x : \sigma . \sigma$. The restriction on type variable x is that it can only have type $\star$.

Types. The definition of types σ is the same. Monotype τ is a special kind of σ that contains no universal quantifiers. Only monotypes can be solutions of unification variables. The well-scopedness of polymorphic type is in Figure 5.

4.2 Subtyping

Declarative Subtyping Unification is only for monotypes. So the definition of unification for the new polymorphic language is the same as Figure 4. However, for a polymorphic language, we need to define the subtyping relationship.

$\boxed{\Gamma \vDash \sigma}$

$$\frac{\Gamma, x : \star \vDash \sigma}{\Gamma \vDash \forall x : \star . \sigma} \text{ WT-Poly}$$

Fig. 5. Well scopedness (extends Figure 1)

$\boxed{\Gamma \vdash \sigma_1 \leq \sigma_2}$

$$\frac{\Gamma, x : \star \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash \sigma_1 \leq \forall x : \star . \sigma_2} \text{ DS-PolyR} \quad \frac{\Gamma \vdash \tau : \star \qquad \Gamma \vdash \sigma_1[x \mapsto \tau] \leq \sigma_2}{\Gamma \vdash \forall x : \star . \sigma_1 \leq \sigma_2} \text{ DS-PolyL}$$

$$\frac{\Gamma \vdash \sigma_3 \leq \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash \sigma_2 \leq \sigma_4}{\Gamma \vdash \Pi x : \sigma_1 .\ \sigma_2 \leq \Pi x : \sigma_3 .\ \sigma_4} \text{ DS-Pi} \quad \frac{}{\Gamma \vdash \sigma \leq \sigma} \text{ DS-AEq}$$

Fig. 6. Declarative Subtyping

The subtyping relationship for polymorphic language is quite standard [4, 6]. We extends it with dependent type in Figure 6. This is a simplified version of subtyping, where we do not consider the polymorphic relationship between constructs other than Pi types and polymorphic types. For example, it is also possible to have subtyping relationship between those two types:

$(\lambda x : Int .\ \forall a . a \to a)\ 3$
$(\lambda x : Int .\ Int \to Int)\ 3$

This simplification is acceptable since our purpose is to deal with unification variables in algorithmic subtyping. Also we expect extending the subtyping with other constructs like application will not cause any problem.

Algorithmic Subtyping For the algorithmic system, we need to add consideration for unification variables in subtyping. Usually, in the case when a unification variable on one side in subtyping process, we will do unification:

$$\frac{\Gamma \vdash \widehat{\alpha} \simeq \sigma \dashv \Theta}{\Gamma \vdash \widehat{\alpha} \sqsubseteq \sigma \dashv \Theta}$$

However, unification only accept monotypes as input types. But we cannot restrict the type in the above rule to be monotype, because there are cases when the type on the other side is a polymorphic type, and the subtyping still holds. For example, in this subtyping

$$\Gamma \vdash \widehat{\alpha} \sqsubseteq \Pi x : (\forall y . y \to y) .\ Int$$

one possible solution for $\widehat{\alpha}$ is $\Pi x : (Int \to Int) .\ Int$.

One observation from this example is that, when the unification variable is on the left, even though there can be polymorphic components on the right hand

$\boxed{\Gamma[\widehat{\alpha}] \vdash^{s} \sigma \rightsquigarrow \tau \dashv \Theta}$ $\quad s = +|-$

$$\frac{\Gamma[\widehat{\alpha}_1, \widehat{\alpha}] \vdash^{+} \sigma[x \mapsto \widehat{\alpha}_1] \rightsquigarrow \tau \dashv \Theta}{\Gamma[\widehat{\alpha}] \vdash^{+} \forall x : \star.\sigma \rightsquigarrow \tau \dashv \Theta}\ \text{I-Poly} \qquad \frac{}{\Gamma \vdash^{s} \tau \rightsquigarrow \tau \dashv \Theta}\ \text{I-Mono}$$

$$\frac{\Gamma \vdash^{-s} \sigma_1 \rightsquigarrow \tau_1 \dashv \Theta_1 \qquad \Theta_1 \vdash^{s} [\Theta_1]\sigma_2 \rightsquigarrow \tau_2 \dashv \Theta}{\Gamma \vdash^{s} \Pi x : \sigma_1.\ \sigma_2 \rightsquigarrow \Pi x : \tau_1.\ \tau_2 \dashv \Theta}\ \text{I-Pi}$$

Fig. 7. Polymorphic Type Sanitization

$\boxed{\Gamma \vdash \sigma_1 \sqsubseteq \sigma_2 \dashv \Theta}$

$$\frac{\Gamma, x : \star \vdash \sigma_1 \sqsubseteq \sigma_2 \dashv \Theta, x : \star, \Delta}{\Gamma \vdash \sigma_1 \sqsubseteq \forall x : \star.\sigma_2 \dashv \Theta}\ \text{S-PolyR} \qquad \frac{\Gamma, \widehat{\alpha} \vdash \sigma_1[x \mapsto \widehat{\alpha}] \sqsubseteq \sigma_2 \dashv \Theta}{\Gamma \vdash \forall x : \star.\sigma_1 \sqsubseteq \sigma_2 \dashv \Theta}\ \text{S-PolyL}$$

$$\frac{\Gamma \vdash \sigma_3 \sqsubseteq \sigma_1 \dashv \Theta_1 \qquad \Theta_1, x : \sigma_1 \vdash [\Theta_1]\sigma_2 \sqsubseteq [\Theta_1]\sigma_4 \dashv \Theta, x : \sigma_1, \Delta}{\Gamma \vdash \Pi x : \sigma_1.\ \sigma_2 \sqsubseteq \Pi x : \sigma_3.\ \sigma_4 \dashv \Theta}\ \text{S-Pi}$$

$$\frac{\Gamma \vdash^{-} \sigma \rightsquigarrow \tau \dashv \Theta_1 \qquad \Theta_1 \vdash \widehat{\alpha} \simeq \tau \dashv \Theta}{\Gamma[\widehat{\alpha}] \vdash \widehat{\alpha} \sqsubseteq \sigma \dashv \Theta}\ \text{S-EVarL} \qquad \frac{\Gamma \vdash^{+} \sigma \rightsquigarrow \tau \dashv \Theta_1 \qquad \Theta_1 \vdash \tau \simeq \widehat{\alpha} \dashv \Theta}{\Gamma[\widehat{\alpha}] \vdash \sigma \sqsubseteq \widehat{\alpha} \dashv \Theta}\ \text{S-EVarR}$$

$$\frac{\Gamma \vdash \sigma_1 \simeq \sigma_2 \dashv \Theta}{\Gamma \vdash \sigma_1 \sqsubseteq \sigma_2 \dashv \Theta}\ \text{S-Unify}$$

Fig. 8. Algorithmic Subtyping

side, those polymorphic components must appear contra-variantly. In the above example, because the type $\forall y.y \to y$ is contra-variant, so it has feasible solutions.

Similar, when the unification variable is on the right, the polymorphic components on the left must appear co-variantly.

Also, if such a solution exists, there could be unlimited number of solutions. For the above example, another solution of $\widehat{\alpha}$ can be $(\Pi x : (\mathit{String} \to \mathit{String}).\ \mathit{Int})$, or $(\Pi x : ((\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int}).\ \mathit{Int})$. Namely, any type of form $\widehat{\alpha}_1 \to \widehat{\alpha}_1$, where $\widehat{\alpha}_1$ is a fresh unification variable. This means, we can eliminate the universal quantifier in the type $(\Pi x : (\forall y.y \to y).\ \mathit{Int})$ by replacing y with a fresh unification variable $\widehat{\alpha}_1$, and then unify $\widehat{\alpha}$ with $(\Pi x : (\widehat{\alpha}_1 \to \widehat{\alpha}_1).\ \mathit{Int})$.

Based on those observations, we give another type sanitization process for polymorphic types in Figure 7. The polymorphic sanitization process will sanitize a probably polymorphic type σ to a monotype τ. This process has two different modes: contra-variant($s = +$), and co-variant($s = -$). Rule I-Poly is when a polymorphic type appears contra-variantly, we eliminate the universal quantifier by replacing x with a fresh unification variable $\widehat{\alpha}_1$. Rule I-Mono is when the type is already a monotype so it remains unchanged. Rule I-Pi switches the mode in this argument type, and preserves the mode in thi return type. Notice there is no rule for a polymorphic type under co-variant mode. For example,

$$\Gamma \vdash \widehat{\alpha} \sqsubseteq \forall y.y \to y$$

will try to sanitize the type $\forall y.y \to y$ under co-variant mode, then it fails, which corresponds to the fact that no solution is feasible for this subtyping.

Having polymorphic type sanitization, we are ready to define algorithmic subtyping relation, which is shown in Figure 8. Rule S-PolyR is intuitive. The trailing context Δ is discarded because the variables are already out of scope. The original non-determinism in rule DS-PolyL is replaced by a new unification variable $\widehat{\alpha}$ in rule S-PolyL. Rule S-Pi also discards useless trailing context. Rule S-EVarL deals with the case a unification variable is on the left, where the type σ is first sanitized co-variantly. Meanwhile, in rule S-EVarR, the unification variable is on the right, and the type σ is sanitized contra-variantly.

4.3 Meta-theory

Conjecture 5. if $\Gamma[\widehat{\alpha}] \vdash^s \sigma \longmapsto \tau \dashv \Theta$, and $\Gamma[\widehat{\alpha}] \vdash \sigma$, then $\Theta \vdash \tau$, and if $s = +$, then $\Theta \vdash \sigma \sqsubseteq \tau \dashv \Theta_2$, else $\Theta \vdash \tau \sqsubseteq \sigma \dashv \Theta_3$.

Proof Sketch. By induction on polymorphic type sanitization rules.

Conjecture 6. if $\Gamma \vdash \sigma_1 \sqsubseteq \sigma_2 \dashv \Theta$, and Δ is an extension of Θ with all unsolved unification variables solved, then $\Delta \vdash [\Delta]\sigma_1 \sqsubseteq [\Delta]\sigma_2$.

Proof Sketch. By induction on subtyping rules.

Conjecture 7. if Θ extends Γ with all unification variables solved, and $[\Theta]\Gamma \vdash [\Theta]\sigma_1 \sqsubseteq [\Theta]\sigma_2$, then there exists a context Θ_2, and Δ', that $\Gamma \vdash [\Gamma]\sigma_1 \sqsubseteq [\Gamma]\sigma_2 \dashv \Theta_2$, and both Θ and Θ_2 can be extended to Δ'.

Proof Sketch. The idea is by induction on declarative subtyping rules, and analyze the form of σ_1, and σ_2. One key observation is that since $[\Theta]\Gamma \vdash [\Theta]\sigma_1 \sqsubseteq [\Theta]\sigma_2$ holds, Θ is itself a feasible solution for the algorithmic subtyping problem $\Gamma \vdash [\Gamma]\sigma_1 \sqsubseteq [\Gamma]\sigma_2 \dashv ?$. However the algorithmic system could have a slightly different solution Θ_2, where some unification variables could remain unsolved.Then both Θ and Θ_2 can extend to a same context Δ.

5 Conclusion

In this paper, we present how to do unification easily for simple dependent types using a strategy called *type sanitization*. This strategy is then extended to deal with subtyping in a language with restricted polymorphic types. In the feature, we plan to finish the sketch proofs for those conjectures. And we are going to extend the algorithm to deal with more features, including system with general polymorphism, higher kinds and so on.

References

[1] Abel, A., Pientka, B.: Higher-order dynamic pattern unification for dependent types and records. In: International Conference on Typed Lambda Calculi and Applications. pp. 10–26. Springer (2011)

[2] Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23(05), 552–593 (2013)

[3] van Doorn, F., Geuvers, H., Wiedijk, F.: Explicit convertibility proofs in pure type systems. In: Proceedings of the Eighth ACM SIGPLAN international workshop on Logical frameworks & meta-languages: theory & practice. pp. 25–36. ACM (2013)

[4] Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: ACM SIGPLAN Notices. vol. 48, pp. 429–442. ACM (2013)

[5] Elliott, C.: Higher-order unification with dependent function types. In: Rewriting Techniques and Applications. pp. 121–136. Springer (1989)

[6] Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of functional programming 17(01), 1–82 (2007)

[7] Kimmell, G., Stump, A., Eades III, H.D., Fu, P., Sheard, T., Weirich, S., Casinghino, C., Sjöberg, V., Collins, N., Ahn, K.Y.: Equational reasoning about programs with general recursion and call-by-value semantics. In: Proceedings of the sixth workshop on Programming languages meets program verification. pp. 15–26. ACM (2012)

[8] Licata, D.R., Harper, R.: A formulation of dependent ml with explicit equality proofs (2005)

[9] McKinna, J.: Why dependent types matter. In: ACM Sigplan Notices. vol. 41, pp. 1–1. ACM (2006)

[10] Norell, U.: Dependently typed programming in agda. In: Advanced Functional Programming, pp. 230–266. Springer (2009)

[11] Pasalic, E., Siek, J., Taha, W.: Concoqtion: Mixing dependent types and hindley-milner type inference (2006)

[12] Sjöberg, V., Casinghino, C., Ahn, K.Y., Collins, N., Eades III, H.D., Fu, P., Kimmell, G., Sheard, T., Stump, A., Weirich, S.: Irrelevance, heterogeneous equality, and call-by-value dependent type systems. arXiv preprint arXiv:1202.2923 (2012)

[13] Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 214–227. ACM (1999)

[14] Ziliani, B., Sozeau, M.: A unification algorithm for coq featuring universe polymorphism and overloading. In: ACM SIGPLAN Notices. vol. 50, pp. 179–191. ACM (2015)

A Appendix

A.1 Typing

Typing is not one of the contributions this paper emphasizes. But because the well-formedness of types under contexts relies on typing, below we give the formalized definition of algorithmic typing used in this paper. Here UV stands for unsolved unification variables.

$\boxed{\Gamma \vdash e : \sigma \dashv \Theta}$

$$\frac{\Gamma \ ctx}{\Gamma \vdash \star : \star \dashv \Gamma}\ \text{T-Ax} \qquad \frac{\Gamma \ ctx \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \dashv \Gamma}\ \text{T-Var} \qquad \frac{\Gamma \ ctx \qquad \widehat{\alpha} \in \Gamma}{\Gamma \vdash \widehat{\alpha} : \star \dashv \Gamma}\ \text{T-EVar}$$

$$\frac{\Gamma \vdash \sigma_1 : \star \dashv \Theta_1 \qquad \Theta_1, x : \sigma_1 \vdash \sigma_2 : \star \dashv \Theta, x : \sigma_1, \Delta}{\Gamma \vdash \Pi x : \sigma_1.\ \sigma_2 : \star \dashv \Theta}\ \text{T-Pi}$$

$$\frac{\Gamma, x \vdash \sigma : \star \dashv \Theta, x, \Delta}{\Gamma \vdash \forall x : \star . \sigma : \star \dashv \Theta}\ \text{T-Poly}$$

$$\frac{\Gamma \vdash \sigma_1 : \star \dashv \Theta_1 \qquad \Theta_1, x : \sigma_1 \vdash e : \tau_2 \dashv \Theta, x : \sigma_1, \Delta}{\Gamma \vdash \lambda x : \tau_1.\ e : (\Pi x : \sigma_1.\ [\Delta]\sigma_2) \dashv \Theta, UV(\Delta)}\ \text{T-LamAnn}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \dashv \Theta_1 \qquad \Theta_1 \vdash_\bullet [\Theta_1]\sigma_1\ e_2 : \sigma_2 \dashv \Theta}{\Gamma \vdash e_1\ e_2 : \sigma_2 \dashv \Theta}\ \text{T-App}$$

$$\frac{\Gamma \vdash e : \sigma_3 \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]\sigma_3 \sqsubseteq [\Theta_1]\sigma_1 \dashv \Theta}{\Gamma \vdash_\bullet (\Pi x : \sigma_1.\ \sigma_2)\ e : \sigma_2[x \mapsto e] \dashv \Theta}\ \text{A-Pi}$$

$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \Pi x : \widehat{\alpha}_1.\ \widehat{\alpha}_2] \vdash e : \sigma \dashv \Theta_1 \qquad \Theta_1 \vdash [\Theta_1]\sigma \sqsubseteq [\Theta_1]\widehat{\alpha}_1 \dashv \Theta}{\Gamma[\widehat{\alpha}] \vdash_\bullet \widehat{\alpha}\ e : \widehat{\alpha}_2 \dashv \Theta}\ \text{A-EVar}$$

$$\frac{\Gamma, \widehat{\alpha} \vdash_\bullet \sigma[\alpha \mapsto \widehat{\alpha}]\ e : \sigma_2 \dashv \Theta}{\Gamma \vdash_\bullet (\forall \alpha : \star . \sigma)\ e : \sigma_2 \dashv \Theta}\ \text{A-Poly}$$

Fig. 9. Typing.

T-Ax, T-Var and T-EVar are standard, where the context has no change.

In T-Pi and T-Poly, the context Δ after x is thrown because all those variables are out of scope, and all the existential variables in Δ will not be refered again when this rule ends. It is safe to throw existential variables because we could assume all unsolved existential variables are solved by $\star$ which satisfys $\star : \star$.

However, in T-LamAnn, because Δ may still contains unsolved unification variables that are referred in σ_2, we keep $UV(\Delta)$ in the output context. But all reference to x is out of scope, so we substitute all solved variables in σ_2.

T-App first infers the type of e_1 to get σ_1, then substitute the context on σ_1 and enter the application typing rules, which is defined by the A-rules. The

judgement $\Gamma \vdash_\bullet \sigma_1\ e : \sigma_2 \dashv \Theta$ is interpreted as, in context Γ, the type of the expression being applied is σ_1, the argument is e_2, and the output of the rule is the application result type σ_2 with context Θ.

In A-Pi, the type of e_1 is a Pi type, so we infer the type of e_2 and check the σ_3 is more polymorphic than σ_1.

In A-EVar, the type of e_1 is an unsolved existential variable $\widehat{\alpha}$. It first destructs $\widehat{\alpha}$ into a Pi type, and infers e_2 to have type σ_1, then check σ_1 is more polymorphic than $\widehat{\alpha}_1$. A-Poly is when the expression being applied has polymorphic type, then we instantiate the bound variable with a fresh unification variable.

A.2 Context Extension

We mentioned some context extends some context several times in the paper informally. Here is the formal definition of the context extension.

$\boxed{\Gamma \longrightarrow \Delta}$

$$\frac{}{\varnothing \longrightarrow \varnothing}\ \text{CE-Empty} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]\sigma_1 = [\Delta]\sigma_2}{\Gamma, x : \sigma_1 \longrightarrow \Delta, x : \sigma_2}\ \text{CE-TypedVar}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, \widehat{\alpha} \longrightarrow \Delta, \widehat{\alpha}}\ \text{CE-EVar} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]\tau_1 = [\Delta]\tau_2}{\Gamma, \widehat{\alpha} = \tau_1 \longrightarrow \Delta, \widehat{\alpha} = \tau_2}\ \text{CE-SolvedEVar}$$

$$\frac{\Gamma \longrightarrow \Delta \qquad \Delta \vdash \tau}{\Gamma, \widehat{\alpha} \longrightarrow \Delta, \widehat{\alpha} = \tau}\ \text{CE-Solve} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \widehat{\alpha}}\ \text{CE-Add} \qquad \frac{\Gamma \longrightarrow \Delta \qquad \Delta \vdash \tau}{\Gamma \longrightarrow \Delta, \widehat{\alpha} = \tau}\ \text{CE-AddSolved}$$

Fig. 10. Context Extension.

All original information will be preserved during context extension, including the existence of variables, their relative orders, type annotations, and solutions for existential. In the meanwhile, information could be increased. Context extension could add more existential variables, or try to solve unsolved existential variables with proper types.

CE-Empty states that empty context could be extended to empty context. CE-TypedVar is able to replace the type annotation with a contextually equivalent one. CE-EVar and CE-SolvedEVar keeps the existential variable. CE-SolvedEVar solves existential variables with a type, requesting the type is well formed. CE-Add and CE-AddSolved adds new existential variable in unsolved and solved form respectively.

It is easy to verify that the output context extends the input context in each relation.