# A Dependently Typed Calculus with Polymorphic Subtyping

Mingqi Xue, Bruno C. d. S. Oliveira

*The University of Hong Kong*

**Abstract**

A *polymorphic subtyping* relation, which relates more general types to more specific ones, is at the core of many modern functional languages. As those languages start moving towards dependently typed programming a natural question is how can polymorphic subtyping be adapted to such settings.

This paper presents the dependent implicitly polymorphic calculus ($\lambda_I^\forall$): a simple dependently typed calculus with polymorphic subtyping. The subtyping relation in $\lambda_I^\forall$ generalizes the well-known polymorphic subtyping relation by Odersky and Läufer (1996). Because $\lambda_I^\forall$ is dependently typed, integrating subtyping in the calculus is non-trivial. To overcome many of the issues arising from integrating subtyping with dependent types, the calculus employs *unified subtyping*, which is a technique that unifies typing and subtyping into a single relation. Moreover, $\lambda_I^\forall$ employs explicit casts instead of a conversion rule, allowing unrestricted recursion to be naturally supported. We prove various non-trivial results, including *type soundness* and *transitivity* of unified subtyping. $\lambda_I^\forall$ and all corresponding proofs are mechanized in the Coq theorem prover.

*Keywords:* Type Systems, Dependent Types, Subtyping, Polymorphism

## 1. Introduction

A *polymorphic subtyping* relation, which relates more general types to more specific ones, is at the core of many modern functional languages. Polymorphic subtyping enables a form of *(implicit) parametric polymorphism*, where type arguments to polymorphic functions are automatically instantiated and the programmer does not specify them. Traditionally, variants of polymorphic subtyping (in the form of a more-general-than relation) have been used in functional languages based on the Hindley-Milner (Hindley, 1969; Milner, 1978; Damas and Milner, 1982) type system, which supports full type-inference without any type annotations. However, the Hindley-Milner type system only supports *rank-1 (or first-order) polymorphism*, where all universal quantifiers only occur at the top-level of a type. Modern functional programming languages, such as Haskell, go beyond Hindley-Milner and support *higher-ranked polymorphism* (Odersky and Läufer, 1996; Peyton Jones et al., 2007) with a more expressive polymorphic

subtyping relation. With higher-ranked polymorphism there is no restriction on where universal quantifiers can occur.

Odersky and Läufer (1996) proposed a simple declarative specification for polymorphic subtyping, which supports higher-ranked polymorphism. Since then several algorithms have been proposed that implement variants of this specification. Most notably, the algorithm proposed by Peyton Jones et al. (2007) forms the basis for the implementation of type inference in the GHC compiler. Dunfield and Krishnaswami (2013) (DK) provide an elegant formalization of another sound and complete algorithm, which has also inspired implementations of type-inference in some polymorphic programming languages, such as PureScript (Freeman, 2017) or DDC (Disciple Development Team, 2017). More recently Zhao et al. (2019) have mechanized DK's type system in a theorem prover.

In recent years dependent types (Coquand and Huet, 1988; Augustsson, 1998; Altenkirch et al., 2010; Sjöberg et al., 2012; Stump et al., 2008; Weirich et al., 2013; Casinghino et al., 2014; Sjöberg and Weirich, 2015) have become a hot topic in programming language research. Several newer functional programming languages, most notably Agda (Norell, 2007) and Idris (Brady, 2013), are now dependently typed. Moreover a number of existing functional languages, such as Haskell, have started to move towards dependently typed programming (Weirich et al., 2017). Dependent types naturally lead to a unification between types and terms, which enables both additional *expressiveness* and *economy of concepts*. The key enabler for unifying terms and types in dependently typed calculi is the adoption of a style similar to Pure Type Systems (PTSs) (Barendregt, 1991). In PTSs there is only a single level of syntax for terms, i.e. the types (or kinds) are expressed using the same syntax as the terms. This is in contrast with more traditional calculi, where distinct pieces of syntax (terms, types and kinds) are separated.

Unified syntax, typical of dependently typed languages, poses some challenges for language design and implementation. A first challenge arises from the interaction between recursion and dependent types. Essentially recursion breaks strong normalization, which many common properties in dependently typed calculi depend upon. One of the most typical properties among them is the decidability of type checking, which simply cannot be guaranteed if some type-level computations are non-terminating. However, this area has been actively investigated in the last few years, and a general approach (Stump et al., 2008; Sjöberg et al., 2012; Kimmell et al., 2012; Sjöberg and Weirich, 2015; Yang et al., 2016) based on explicit casts for type-level computations, has emerged as an interesting solution for integrating general recursion in dependently typed calculi. By avoiding the implicit type-level computation entirely, whether programs strongly normalize or not no longer matters for the decidability of type checking. Current proposals for dependently typed versions of Haskell (Weirich et al., 2017), for instance, adopt explicit casts for type-level computation.

The second challenge, for calculi that employ subtyping, is that smoothly integrating dependent types and subtyping is difficult. Subtyping is a substantial difference to traditional PTSs, which do not have this feature. The issue

with subtyping is well summarized by Aspinall and Compagnoni (1996): *"One thing that makes the study of these systems difficult is that with dependent types, the typing and subtyping relations become intimately tangled, which means that tested techniques of examining subtyping in isolation no longer apply"*. Recent work on *unified subtyping* (Yang and Oliveira, 2017) provides a simple technique to address this problem. Following the same spirit as Pure Type Systems, which attempt to unify syntax and the typing and well-formedness relations, unified subtyping suggests unifying typing and subtyping into a single relation. This solves the problem of dependencies in that now there is only a single relation that depends only on itself. Furthermore, it results in a compact specification compared to a variant with multiple independent relations.

In this paper, we investigate how polymorphic subtyping can be adapted into a dependently typed calculus with general recursion and explicit casts for type-level computation. We employ unified subtyping to address the issues of combining dependent types with subtyping. The use of explicit casts for type-level computation means that type equality is essentially syntactic (or rather up-to $\alpha$-equivalence). This avoids the use of a traditional conversion rule that allows concluding $\beta$-equivalent types to be equal. In essence, the use of the conversion rule requires (implicit) type-level computation, since terms have to be normalized using $\beta$-reduction to conclude whether or not they are equal. Dependent type systems with a conversion rule have some major complications. A well-known one is that type-inference for such systems requires *higher-order unification*, which is known to be *undecidable* (Goldfarb, 1981). By employing a system with $\alpha$-equivalence only we stay closer to existing languages like Haskell, where type equality (at least at the core language level) is also essentially only up-to $\alpha$-equivalence.

We present a calculus called $\lambda_I^\forall$, and show three main results in this paper: *transitivity of subtyping*, *type soundness*, and *completeness of $\lambda_I^\forall$'s polymorphic subtyping with respect to Odersky and Läufer's formulation*. Transitivity is a non-trivial result (like in most calculi combining dependent types and subtyping) and requires a proof based on sizes and a property that guarantees the uniqueness of kinds in our language. Type soundness is also non-trivial and we need to take a different approach than that employed by existing work on polymorphic subtyping (Odersky and Läufer, 1996; Peyton Jones et al., 2007), where type-safety is shown by an elaboration to System F. In essence elaboration into a target language brings significant complications to the metatheory in a dependently typed setting. Thus, instead of elaboration, we use a direct operational semantics approach, which is partly inspired by the approach used in the *Implicit Calculus of Constructions* (ICC) (Miquel, 2001; Barras and Bernardo, 2008), to prove type soundness. Similar to ICC, we adopt the restriction that arguments for implicit function types are computationally irrelevant (i.e. they cannot be used in runtime computation). However, our unified subtyping setting is significantly different from ICC due to the presence of subtyping, which brings complications the ICC does not have. We also prove that any valid subtyping statement in the Odersky and Läufer relation is valid in $\lambda_I^\forall$. Thus $\lambda_I^\forall$'s unified subtyping subsumes the polymorphic subtyping relation by Odersky and

Läufer.

$\lambda_I^\forall$ and all the proofs reported in this paper are formalized in the Coq theorem prover (Coq development team). This paper does not address decidability or soundness and completeness of $\lambda_I^\forall$ to an algorithmic formulation, which are outside of the scope of this work. Nonetheless, these are important and challenging questions for practical implementations of $\lambda_I^\forall$, which are left open for future work.

In summary, the contributions of this paper are:

- **The $\lambda_I^\forall$ calculus**, which is a dependently typed calculus with explicit casts, general recursion and implicit higher-ranked polymorphism.
- **Type-soundness and transitivity of subtyping.** We show that $\lambda_I^\forall$ is type-sound and unified subtyping is transitive.
- **Subsumption of Odersky and Läufer's polymorphic subtyping.** We show that $\lambda_I^\forall$'s unified subtyping can encode all valid polymorphic subtyping statements of Odersky and Läufer's relation.
- **Mechanical formalization.** All the results have been mechanically formalized in the Coq theorem prover. The formalization is available online at: `https://github.com/VinaLx/dependent-polymorphic-subtyping`.

## 2. Overview

In this section, we introduce $\lambda_I^\forall$ by going through some interesting examples to show the expressiveness and major features of the calculus. Then we discuss the motivation, rationale of our design, and challenges. The formal system of $\lambda_I^\forall$ will be discussed in Sections 3 and 4.

### *2.1. A Tour of $\lambda_I^\forall$*

The $\lambda_I^\forall$ calculus features a form of *implicit higher-ranked polymorphism* with the power of dependent types. Thus the main feature of $\lambda_I^\forall$ is the ability to implicitly infer universally quantified arguments.

*A First Example of Implicit Polymorphism.* Like most of functional languages, $\lambda_I^\forall$ supports (implicit) parametric polymorphism. The first simple example is the polymorphic identity function:

```
id : ∀(A:⋆). A → A
id = λ(x:A). x
answer : Nat
answer = id 42 -- No type argument needed at the call of id
```

The polymorphic parameter `A` is annotated with its type, which is $\star$. The type $\star$ is the type of types (also known as *kind*). In $\lambda_I^\forall$, the parameters of lambda abstractions are annotated with their types, and the `A` in the definition refers back to the polymorphic parameter. In the examples below, we drop the

parentheses around variables and their type annotations such as `λx:A. x` for conciseness.

Similar to implicit polymorphism in other languages, the polymorphic parameters of the ∀ types are implicitly instantiated during applications. Thus, in the call of the identity function (`id 42`), we do not need to specify the argument used for instantiation. In contrast, in an explicitly polymorphic language (such as System F) we would need to call `id` with an extra argument that specifies the instantiation of `A`: `id Nat 42`.

*Recursion and Dependent Types.* $\lambda_I^{\forall}$ is dependently typed, and universal quantifications are not limited to work on arguments of type $\star$, but it allows arguments of other types. This is a key difference compared to much of the work on type-inference for higher-ranked polymorphism (Dunfield and Krishnaswami, 2013; Le Botlan and Rémy, 2003; Leijen, 2008; Vytiniotis et al., 2008; Peyton Jones et al., 2007) which has been focusing on System F-like languages where universal quantification can only have arguments of type $\star$. Furthermore, $\lambda_I^{\forall}$ supports general recursion at both the term and the type-level.

Using these features we can encode an indexed list, a `map` operation over it , and we illustrate how the implicit instantiation allows us to use the `map` function conveniently. However, because $\lambda_I^{\forall}$ is just a core calculus there is no built-in support yet for algebraic datatypes and pattern matching. We expect that a source language would provide a more convenient way to define the `map` function using pattern matching and other useful source-level constructs. To model algebraic datatypes and pattern matching in $\lambda_I^{\forall}$, we use an encoding by Yang and Oliveira (Yang and Oliveira, 2019), which is based on the Scott Encodings (Mogensen, 1992). The Scott Encoding encodes datatypes with different cases via Continuation-Passing-Style (CPS) function types. The return branches of these function types correspond to each case of the datatypes. Case analyses of terms are encoded via applications of the CPS functions. Since the details of the encoding are not relevant to this paper, here we omit the code for most definitions and show only their types.

In a dependently typed language a programmer could write the following definition for our formulation of indexed lists:

```
data Nat = Zero | Succ Nat
data List (a : ⋆) (n : Nat) = Nil | Cons a (List a (Succ n))
```

In this definition, the index grows towards the tail of the list, which is admittedly not the most useful definition. The reason why we did not choose the more practical example, where the index represents the length of the list, is that it requires encodings of GADT-like datatypes (Cheney and Hinze, 2003; Xi et al., 2003). Such encodings are more complex than encodings of regular algebraic datatypes as they require explicit equality proofs and more language-level supports for such proofs (Yang and Oliveira, 2019). Thus we use the simpler, but still dependently typed example here. Here we encode `List` and its constructors as conventional terms. We show the definition for `List`, and the types for the constructors next (implementation omitted):

```
List : ⋆ → Nat → ⋆
List = μL:(⋆ → Nat → ⋆).λa:⋆.λn:Nat.Πr:⋆.
         r                          -- Nil branch
       → (a → L a (Succ n) → r)     -- Cons branch
       → r
Nil  : ∀a:⋆.∀n:Nat. List a n
Cons : ∀a:⋆.∀n:Nat. a → List a (Succ n) → List a n
```

In subsequent examples we will just assume some Haskell-style syntactic sugar for datatype definitions and constructors. Using the definitions above, we can define a `map` function over `List` with the type:

```
map : ∀a:⋆.∀b:⋆.∀n:Nat. a → b → List a n → List b n
```

An example of application of `map` is:

```
map Succ (Cons 1 (Cons 2 Nil))
```

which increases every natural in the list by one. Note that since the type parameters for `map`, `Cons`, and `Nil` are all implicit, they can be all omitted and the arguments are instantiated implicitly. Thus the `map` function only requires two explicit arguments, making it as convenient to use as in most functional language implementations.

There are a few final points worth mentioning about the example. Firstly, `List` is an example of a dependently typed function, since it is parameterized by a natural value. Secondly, following the design of PITS (Yang and Oliveira, 2019), fixpoint operators ($\mu$) in $\lambda_I^{\forall}$ serve a dual purpose of defining recursive types and recursive functions. Besides its usual use of defining term-level general recursive functions, fixpoint operators can be used to define recursive types, as shown in the encoding of `List` above. Moreover, recursion is unrestricted and there is no termination checking, much like approaches such as Dependently Typed Haskell (Eisenberg, 2016), and unlike various other dependently typed languages such as Agda (Norell, 2007) or Idris (Brady, 2013).

*Implicit Higher-Kinded Types.* The implicit capabilities also extend to the realm of higher-kinded types (Pierce, 2002). For example, we can define a record type `Functor`, to mimic the type class (Wadler and Blott, 1989; Kaes, 1988) `Functor` in Haskell:

```
data Functor (F : ⋆ → ⋆) =
  MkF { fmap : ∀a:⋆.∀b:⋆. (a → b) → F a → F b }
```

Here Functor is a record type with a single field. MkF is the data constructor, and `fmap` is the field accessor (which encodes the type class method `fmap`). The type of `fmap` is:

```
fmap : ∀F:⋆ → ⋆. Functor F → (∀a:⋆.∀b:⋆. (a → b) → F a → F b)
```

Importantly this example illustrates that universal variables can quantify over higher-kinds (i.e. `F : ⋆ → ⋆`). We can define instances of functor in a standard way:

```
data Id a = MkId { runId : a }
idFunctor : Functor Id
idFunctor =
  MkF { fmap = λf:(a → b).λx:(Id a). MkId (f (runId x)) }
```

and then use `fmap` with three arguments:

```
fmap idFunctor Succ (MkId 0)
```

Note that, because our calculus has no mechanism like type classes we pass the "instance" explicitly. Nonetheless, three other arguments (the `F`, `a`, and `b`) are implicitly instantiated.

*Higher-Ranked Polymorphic Subtyping.* In calculi such as the ICC (Miquel, 2001), a form of implicit instantiation also exists. However, such calculi do not employ subtyping, instead, they only apply instantiation to top-level universal quantifiers. Our next example illustrates how subtyping enables instantiation to be applied also in nested universal quantifiers, thus enabling more types to be related.

When programming with continuations (Sussman and Steele, 1998) one of the functions that are typically needed is call-with-current-continuation (`callcc`). In a polymorphic language, there are several types that can be assigned to `callcc`. One of these types is a rank-3 type, while another one is a rank-1 type. Using polymorphic subtyping we can show that the rank-3 type is more general than the rank-1 type. Thus the following program type-checks:

```
callcc' : ∀a:⋆. ((∀b:⋆. a → b) → a) → a
callcc : ∀a:⋆.∀b:⋆. ((a → b) → a) → a
callcc = callcc'
```

The type `∀b:⋆. a → b` appears in a positive position of the whole signature, and it is a more general signature than `a → b` for an arbitrary choice of `b`. Our language captures this subtyping relation so that we can assign `callcc'` to `callcc` (but not the other way around). In contrast, in approaches like the ICC, the types of `callcc` and `callcc'` are not compatible and the example above would be rejected.

### *2.2. Key Features*

We briefly discuss the major features of $\lambda_I^{\forall}$ itself and its formalization. More formal and technical discussions will be left to Sections 3 and 4.

*Polymorphic Subtyping Relation.* Figure 1 shows the syntax of types, monomorphic types (or monotypes), and the polymorphic subtyping relation in Dunfield and Krishnaswami (2013) variation of Odersky and Läufer's declarative type system (Odersky and Läufer, 1996). Although there are slight differences between the two versions of subtyping relations, since they essentially express the same idea, we use DK's and Odersky and Läufer's polymorphic subtyping relation interchangeably in this article. Here the syntax includes ∀ types that

$$
\begin{array}{llll}
\text{Types} & A, B & ::= & x \mid \mathbb{N} \mid A \to B \mid \forall x.\, A \\
\text{Monotypes} & \tau, \sigma & ::= & x \mid \mathbb{N} \mid \tau \to \sigma
\end{array}
$$

$\boxed{\Gamma \vdash_{\text{DK}} A \leq B}$ *(Polymorphic Subtyping)*

$$
\frac{x \in \Gamma}{\Gamma \vdash_{\text{DK}} x \leq x}\ \leq\text{Var}
\qquad
\frac{}{\Gamma \vdash_{\text{DK}} \mathbb{N} \leq \mathbb{N}}\ \leq\text{Int}
\qquad
\frac{\Gamma \vdash_{\text{DK}} B_1 \leq A_1 \qquad \Gamma \vdash_{\text{DK}} A_2 \leq B_2}{\Gamma \vdash_{\text{DK}} A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to
$$

$$
\frac{\Gamma \vdash_{\text{DK}} \tau \qquad \Gamma \vdash_{\text{DK}} [\tau/x]\, A \leq B}{\Gamma \vdash_{\text{DK}} \forall x.\, A \leq B}\ \leq\forall L
\qquad
\frac{\Gamma, x \vdash_{\text{DK}} A \leq B}{\Gamma \vdash_{\text{DK}} A \leq \forall x.\, B}\ \leq\forall R
$$

Figure 1: The Dunfield and Krishnaswami (2013) variation of the polymorphic subtyping relation by Odersky and Läufer (1996).

represent polymorphic types (or polytypes), which are universally quantified over type parameters. The definition of monotypes is standard and includes all types without occurrences of universal quantifiers. Context $\Gamma$ is a list of type variables that are allowed to occur free in types $A$ and $B$ in the subtyping relation. The polymorphic subtyping relation captures a *more-general-than* relation between types. The key rules in their subtyping relation are rules $\leq\forall L$ and $\leq\forall R$:

- In rule $\leq\forall L$, a polytype $(\forall x.\, A)$ is considered *more-general* than another type $(B)$, when we can find an arbitrary monotype $(\tau)$ so that the instantiation is more general than $B$. Importantly note that this relation does not guess arbitrary (poly)types, but just monotypes. In other words, the relation is *predicative* (Martin-Löf, 1998). This restriction ensures that the relation is *decidable*.
- In rule $\leq\forall R$ a type $(A)$ is considered more general than a polytype $(\forall x.\, B)$ when it is still more general than the head of the polytype, with the type parameter instantiated by an abstract variable $x$.

This subtyping relation sets a scene for our work, which generalizes this relation to a dependently typed setting.

*Generalizing Polymorphic Subtyping.* The parameters of universal types can only be types in the polymorphic subtyping relation by Odersky and Läufer. In $\lambda_I^{\forall}$, we generalize the polymorphic parameters so that they can be values or other kinds of types as well. The first idea for a direct generalization is:

$$\frac{\Gamma \vdash \tau : A \qquad \Gamma \vdash [\tau/x]\, B \leq C}{\Gamma \vdash \forall x : A \,.\, B \leq C}\ {}^{\leq \forall L'} \qquad \frac{\Gamma,\, x : B \vdash A \leq C}{\Gamma \vdash A \leq \forall x : B \,.\, C}\ {}^{\leq \forall R'}$$

The parameters for universal types can have any type (and not just $\star$). Hence, instead of requiring the monotype $\tau$ to be a well-formed type in rule $\leq\forall L$, in rule $\leq\forall L'$ it is required that $\tau$ is well-typed regarding the type of the parameter in the universal quantifier. Similarly, for rule $\leq\forall R'$ the context for the subtyping rule should include typing information for the universally quantified variable. However, this idea introduces the issue of potential mutual dependency between subtyping and typing judgments, so further adjustments have to be made to formalize this idea, which is discussed later in this section, Sections 3.3 and 4.1.

*Higher-Ranked Polymorphic Subtyping.* As the `callcc` example in Section 2.1 shows, the subtyping rules based on polymorphic subtyping, combined with other subtyping rules, are able to handle the subtyping relations that occur at not only top-level but also at a higher-ranked level. This feature distinguishes our $\lambda_I^{\forall}$ from the Implicit Calculus of Constructions (ICC) (Miquel, 2001) which also discusses the implicit polymorphism of dependent type languages. The ICC features these two related rules in their *typing* relation:

$$\frac{\Gamma \vdash e : \forall\, x : A.\ B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e : [e_1/x]\, B}\ {}^{\text{INST}} \qquad \frac{\Gamma,\, x : A \vdash e : B \qquad \Gamma \vdash \forall\, x : A.\ B : k}{\Gamma \vdash e : \forall\, x : A.\ B}\ {}^{\text{GEN}}$$

Without an explicit subtyping relation, the ICC is not always able to handle subtyping at higher-ranked positions. The approach taken by the ICC is similar to that of the Hindley-Milner type system (Hindley, 1969; Damas and Milner, 1982), which is also designed for dealing only with rank-1 polymorphism. Hindley-Milner's declarative system also has a GEN rule to convert expressions to polymorphic types, and an INST rule to instantiate polymorphic parameters. Similar to rules GEN and INST shown above, both rules in HM work only for polymorphic types at top-level positions. In Hindley-Milner the universal quantifier can only quantify over types, whereas in the ICC it can quantify over terms of an arbitrary type (including types themselves). In generalizations of higher-ranked polymorphic type-inference (Dunfield and Krishnaswami, 2013; Le Botlan and Rémy, 2003; Leijen, 2008; Vytiniotis et al., 2008; Peyton Jones et al., 2007), it has been shown that rules like $\leq\forall L$ and $\leq\forall R$ generalize rules like GEN and INST. Since we aim at higher-ranked polymorphic generalization, we follow a similar, more general, approach in $\lambda_I^{\forall}$.

*Unified Subtyping.* The revised subtyping relation with $\leq\forall L'$ and $\leq\forall R'$ rules suffers from an important complication compared to the Odersky and Läufer formulation: there is now a notorious mutual dependency between typing and

subtyping. In Odersky and Läufer's rules, the subtyping rules do not depend on typing. In particular the rule $\leq\forall L$ depends only on well-formedness ($\Gamma \vdash \tau$). In contrast, note that rule $\leq\forall L'$ now mentions the typing relation in its premise ($\Gamma \vdash \tau : A$). Moreover, as usual, the subsumption rule of the typing relation depends on the subtyping relation as shown below.

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\ \text{T-SUB}$$

This mutual dependency has been a significant problem when combining subtyping and dependent types (Aspinall and Compagnoni, 1996; Hutchins, 2010), and presents itself on our way to the direct generalization of polymorphic subtyping.

To tackle this issue, we adopt a technique called the *unified subtyping* (Yang and Oliveira, 2017). Unified subtyping merges the typing relation and subtyping relation into a single relation to avoid this mutual dependency:

$$\Gamma \vdash e_1 \leq e_2 : A$$

The interpretation of this judgment is: under context $\Gamma$, $e_1$ is a subtype of $e_2$ and they both are of type $A$. The judgments for subtyping and typing are both special forms of unified subtyping:

$$\Gamma \vdash A \leq B \triangleq \Gamma \vdash A \leq B : \star \qquad\qquad \Gamma \vdash e : A \triangleq \Gamma \vdash e \leq e : A$$

The technique simplifies the formalization of dependently typed calculi with subtyping, and especially the proof of transitivity in the original work by Yang and Oliveira (2017). After applying the technique, an ideal generalization of the polymorphic subtyping would be:

$$\frac{\Gamma \vdash \tau : A \qquad \Gamma \vdash [\tau/x]\, B \leq C : \star}{\Gamma \vdash \forall x : A.\, B \leq C : \star}\ \leq\forall L'' \qquad\qquad \frac{\Gamma,\, x : B \vdash A \leq C : \star}{\Gamma \vdash A \leq \forall x : B.\, C : \star}\ \leq\forall R''$$

The basic idea of our own formalization essentially follows a similar design, although the actual rules in $\lambda_I^{\forall}$ are slightly more sophisticated. The details will be discussed in Section 3.3.

*"Explicit" Implicit Instantiation.* With polymorphic subtyping the instantiation of universally quantified type parameters is done implicitly instead of being manually applied. In non-dependent type systems, *implicit* parameters are types (i.e. terms are not involved in implicit instantiation). For example:

$$(\lambda x.\, x)\ 42 \longrightarrow 42$$

Here $\lambda x.\, x$ has type $\forall A.\, A \to A$, and instantiation implicitly discovers that $A$ = Int. Notably, and in contrast with explicitly polymorphic languages like System

F, implicit instantiation is not reflected anywhere at term level. The design that we adopt still provides implicit instantiation, but it is more explicit regarding the binding of implicit parameters. We adopt this design to ensure that polymorphic variables are well-scoped in type annotations of terms. Thus we use another binder, of the form $\Lambda(x : A).e$, for terms. Nonetheless, instantiations are still implicit as shown in the following example:

$$(\Lambda A : \star.\, \lambda x : A.\, x)\ 42 \longrightarrow 42$$

Here $\Lambda A : \star.\, \lambda x : A.\, x$ has type $\forall A : \star.\, A \to A$, and the polymorphic parameter $A$ is explicitly stated in the polymorphic term. However as the reduction shows, the instantiations are still implicit. We purposely omitted the explicit binders for implicit parameters for all the examples in Section 2.1 for conciseness. Such explicit binders can be recovered with a simple form of syntactic sugar:

$$e : \forall (x : A).\, B \triangleq \Lambda x : A.\, e : \forall (x : A).\, B$$

*Computational Irrelevance.* Implicit parameters in traditional languages with polymorphic subtyping, the ICC (Miquel, 2001; Barras and Bernardo, 2008), and $\lambda_I^{\forall}$ are computationally irrelevant. In traditional (non-dependently) typed languages, types cannot affect computation, thus computational irrelevance is quite natural and widely adopted. Furthermore, computational irrelevance can benefit performance, since irrelevant arguments can simply be erased at runtime. In dependently typed systems, however, there can be some programs where it is useful to have computationally relevant implicit parameters. For example, accessing the length of a length-indexed vector in constant time:

```
length : ∀n:Nat. Vector n → Nat
length = Λn:Nat.λv:(Vector n). n
```

Here the implicit parameter `n` is computationally relevant as it is used as the return value of the function which is likely to be executed at runtime. Languages like Agda, Coq, and Idris support such programs. However, computationally relevant implicit parameters are challenging for proofs of type soundness. Due to such challenges (see also the discussion in Section 3.2), the ICC has a restriction that parameters for implicit function types must be computationally irrelevant. Since we adopt a similar technique for the type soundness proof, we also have a similar restriction and thus cannot encode programs such as the above.

*Type-level Computation and Casts.* $\lambda_I^{\forall}$ features the fixpoint operator that supports general recursion at both type and term level. In order to avoid diverging computations at type checking, we do not provide the conversion rule (or congruence rule) like other dependently typed systems such as the Calculus of Constructions (Coquand and Huet, 1988) to support implicit type-level reduc-

tion.

$$\textsc{Cong}\ \frac{\Gamma \vdash e : A \qquad A =_\beta B}{\Gamma \vdash e : B}$$

The presence of the conversion rule makes the decidability of type checking rely on the strong normalization of type-level computation (to determine whether two types are $\beta$-equivalent). But the presence of general recursion denies the strong normalization property of our language.

Instead of using a conversion rule, we adopt the call-by-name design of *Pure Iso-Type Systems* (PITS) (Yang et al., 2016; Yang and Oliveira, 2019), and provide $\text{cast}_{\Downarrow}$ and $\text{cast}_{\Uparrow}$ operators to explicitly trigger one-step type reductions or expansions as shown in the typing rules below.

$$\textsc{Castup}\ \frac{\Gamma \vdash e : B \qquad A \longrightarrow B \qquad \Gamma \vdash A : k}{\Gamma \vdash \text{cast}_{\Uparrow}\,[A]\,e : A} \qquad \textsc{Castdn}\ \frac{\Gamma \vdash e : A \qquad A \longrightarrow B \qquad \Gamma \vdash B : k}{\Gamma \vdash \text{cast}_{\Downarrow}\,e : B}$$

Now, since reductions only perform one step per use of cast operators, whether a term strongly normalizes or not no longer affects the decidability of type checking. Note that there are some other cast designs in the literature (Stump et al., 2008; Sjöberg et al., 2012; Kimmell et al., 2012; Sjöberg and Weirich, 2015), we adopt the PITS design here for simplicity. We believe that other cast designs could also be adopted instead, but leave this for future work.

## 3. The Dependent Implicitly Polymorphic Calculus

This section introduces the static and dynamic semantics of $\lambda_I^\forall$: a dependently typed calculus with type casts and implicit polymorphism. The calculus employs *unified subtyping* (Yang and Oliveira, 2017) and has a single relation that generalizes both typing and subtyping. The calculus can be seen as a variant of the *Calculus of Constructions* (Coquand and Huet, 1988), but it uses a simple form of casts (Yang et al., 2016; Yang and Oliveira, 2019) with $\text{cast}_{\Uparrow}$ and $\text{cast}_{\Downarrow}$ operators instead of the conversion rule and features unrestricted recursion with the fixpoint operator. We present the syntax, the unified subtyping relation, and operational semantics for $\lambda_I^\forall$.

### *3.1. Syntax*

Figure 2 shows the syntax of $\lambda_I^\forall$. The syntax is similar to the Calculus of Constructions, featuring unified terms and types, and a kind hierarchy with $\star$ and $\Box$. The kind $\star$ is the type (or kind) of other types like $\mathbb{N}$ and $\Pi$ types, the kind $\Box$ is the type of $\star$, but $\Box$ itself has no type/kind. Due to the unified syntax, types and expressions ($e$, $A$ and $B$) are used interchangeably, but we mostly adopt the convention of using $A$ and $B$ for contexts where the expressions are used as types and $e$ for contexts where the expressions represent terms. The

| | | | |
|---|---|---|---|
| Kinds | $k$ | $::=$ | $\star \mid \Box$ |
| Expressions | $e, A, B$ | $::=$ | $x \mid n \mid k \mid \mathbb{N} \mid e_1\ e_2 \mid \lambda\, x : A.\ e \mid \Pi\, x : A.\ B$ |
| | | $\mid$ | $\Lambda\, x : A.\ e \mid \forall\, x : A.\ B \mid \mu\, x : A.\ e$ |
| | | $\mid$ | $\mathrm{cast}_{\Uparrow}\, [A]\, e \mid \mathrm{cast}_{\Downarrow}\, e$ |
| Mono-Expressions | $\tau, \sigma$ | $::=$ | $x \mid n \mid k \mid \mathbb{N} \mid \tau_1\ \tau_2 \mid \lambda\, x : \tau.\ \sigma \mid \Pi\, x : \tau.\ \sigma$ |
| | | $\mid$ | $\Lambda\, x : \tau.\ \sigma \mid \mu\, x : \tau.\ \sigma \mid \mathrm{cast}_{\Uparrow}\, [\tau]\ \sigma \mid \mathrm{cast}_{\Downarrow}\, \tau$ |
| Values | $v$ | $::=$ | $k \mid n \mid \mathbb{N} \mid \lambda\, x : A.\ e \mid \Pi\, x : A.\ B \mid \Lambda\, x : A.\ e$ |
| | | $\mid$ | $\forall\, x : A.\ B \mid \mathrm{cast}_{\Uparrow}\, [A]\, e$ |
| Contexts | $\Gamma$ | $::=$ | $\varnothing \mid \Gamma,\, x : A$ |
| Syntactic Sugar | $A \to B$ | $\triangleq$ | $\Pi\, x : A.\ B \qquad$ where $x \notin \mathrm{FV}(B)$ |

Figure 2: Syntax of $\lambda_I^{\forall}$.

syntax includes all the constructs of the calculus of constructions: variables ($x$), kinds ($k$), function applications ($e_1\ e_2$), lambda expressions ($\lambda\, x : A.\ e$), dependent function types ($\Pi\, x : A.\ B$) as well as integer types ($\mathbb{N}$) and integers ($n$). Moreover, there are several additional language constructs to support implicit polymorphism, recursion, and explicit type-level computation via casts. These constructs are discussed next.

*Implicit Polymorphism.* In $\lambda_I^{\forall}$, universal types $\forall\, x : A.\ B$ generalize implicit polymorphic types ($\forall x.\, A$) in conventional functional languages. The parameter $x$ in $\forall\, x : A.\ B$ ranges over all well-typed expressions besides well-formed types (i.e. $x$ can have any type $A$ instead of just kind $\star$). The idea of *monomorphic types* (or monotypes) is also generalized. *Mono-expressions* $\tau$ exclude polymorphic types ($\forall\, x : A.\ B$) from the syntax, which follows a similar design in various work about predicative higher-ranked polymorphism (Odersky and Läufer, 1996; Dunfield and Krishnaswami, 2013; Peyton Jones et al., 2007). With dependent types, only generalized universal types are excluded, but not any other related expressions.

Notably, generalized "polymorphic types" are naturally dependent, and $\forall$ types can be viewed as the implicit counterpart of $\Pi$ types. So we also have implicit lambda expressions ($\Lambda\, x : A.\ e$), which is different from the "usual" $\lambda$ expressions and explicit function types ($\Pi$ types), for which arguments should be explicitly passed. The arguments of implicit lambda ($\Lambda$) expressions are deduced during type checking. This is a similar design to the *Implicit Calculus of Constructions* (ICC$^*$) (Barras and Bernardo, 2008), which employs similar constructs for implicit dependent products.

*Recursion and Explicit Type-level Computation.* The $\lambda_I^{\forall}$ calculus adopts *iso-types* (Yang et al., 2016; Yang and Oliveira, 2019), featuring explicit type-level computation with cast operators $\mathrm{cast}_{\Downarrow}$ and $\mathrm{cast}_{\Uparrow}$. These operators respectively perform one-step type *reduction* and *expansion* based on the operational semantics. The reduction in cast operators is deterministic, thus type annotations are

only needed during type expansions ($\text{cast}_{\Uparrow}$). We add fixpoints ($\mu\, x : A.\ e$) to support general recursion for both term-level and type-level. Iso-recursive types are supported by $\text{cast}_{\Uparrow}$ and $\text{cast}_{\Downarrow}$ operators, which correspond to the `fold` and `unfold` operations when working on conventional iso-recursive types.

### *3.2. Operational Semantics*

For the operational semantics we employ two different, but closely related reduction relations. The first reduction relation is non-deterministic, and it is used at the type-level to allow type conversions induced by the cast operators. The second reduction relation is deterministic and is employed to give the runtime semantics of expressions.

*Non-deterministic Implicit Instantiations.* Figure 3 presents the small-step operational semantics of $\lambda_I^{\forall}$. It mostly follows the "Call-By-Name" (CBN) variant of *Pure Iso-Type Systems* (PITS) (Yang and Oliveira, 2019) corresponding to the calculus of constructions. In such variant the arguments of the $\beta$-reduction (rule R-BETA) and expressions in the rule R-CAST-ELIM are not required to be values. Reductions can be performed inside $\text{cast}_{\Downarrow}$ terms (rules R-CASTDN and R-CAST-INST). Note that here rule R-CASTDN reflects the term-level reduction of $\text{cast}_{\Downarrow}$ terms themselves, the one-step type-level reductions triggered by the $\text{cast}_{\Downarrow}$ operator are reflected in the typing rules. Following the CBN semantics of PITS, $\text{cast}_{\Uparrow}$ terms are considered to be values to avoid nondeterministic reduction of terms like $\text{cast}_{\Downarrow}(\text{cast}_{\Uparrow}[A]\ e)$, where $e$ is reducible (either performing reduction on $e$, or the reduction via rule R-CAST-ELIM could be the choice, should $\text{cast}_{\Uparrow}$ terms be reducible). There is an alternative design following the "call-by-value" variant of PITS, which we will discuss in Section 5.2. Also, the unfold operation of the fixpoint operator is supported by rule R-MU.

Due to the presence of instantiation of implicit parameters, the direct operational semantics is not deterministic, and potentially not type-preserving because of rules R-INST and R-CAST-INST. The indeterminacy is caused by the guess of $\tau$, which can be an arbitrary mono-expression, since we do not have access to any typing information in the dynamic semantics.

*Deterministic Erased Reduction.* We address the issue of determinacy of the dynamic semantics with a design similar to ICC$^*$ (Barras and Bernardo, 2008), employing type-erased expressions. The erased expressions essentially mirror the syntax and semantics of normal expressions, except for the elimination of type annotations in $\lambda$, $\Lambda$, $\mu$, and $\text{cast}_{\Uparrow}$ expressions. Figure 4 shows the syntax of the erased expressions and the companion operational semantics. Restrictions are imposed in the typing rules to forbid the implicit parameter occurring in runtime-relevant parts of the expression, i.e. the erased expressions (see Section 3.3). With such restriction, implicit parameters can be directly eliminated in rules ER-ELIM and ER-CAST-INST. For a well-typed expression, the reduction of its erasure is deterministic. Although the implicit parameter does not matter at runtime, the erasure function preserves the structure of the original syntax by not eliminating the implicit binder altogether. This design has the advantage

$\boxed{e_1 \longrightarrow e_2}$ *(Operational Semantics)*

$$\textsc{r-app}\ \frac{e_1 \longrightarrow e_2}{e_1\ e_3 \longrightarrow e_2\ e_3} \qquad \textsc{r-beta}\ \frac{}{(\lambda\, x : A.\ e_1)\ e_2 \longrightarrow [e_2/x]\, e_1}$$

$$\textsc{r-inst}\ \frac{}{(\Lambda\, x : A.\ e_1)\ e_2 \longrightarrow ([\tau/x]\, e_1)\ e_2} \qquad \textsc{r-mu}\ \frac{}{\mu\, x : A.\ e \longrightarrow [\mu\, x : A.\ e/x]\, e}$$

$$\textsc{r-castdn}\ \frac{e_1 \longrightarrow e_2}{\mathrm{cast}_{\Downarrow}\ e_1 \longrightarrow \mathrm{cast}_{\Downarrow}\ e_2} \qquad \textsc{r-cast-inst}\ \frac{}{\mathrm{cast}_{\Downarrow}\ (\Lambda\, x : A.\ e) \longrightarrow \mathrm{cast}_{\Downarrow}\ ([\tau/x]\, e)}$$

$$\textsc{r-cast-elim}\ \frac{}{\mathrm{cast}_{\Downarrow}\ (\mathrm{cast}_{\Uparrow}\ [B]\, e) \longrightarrow e}$$

Figure 3: Operational semantics of $\lambda_I^{\forall}$.

that the correspondence between the original and the erased expression can be established more easily, when the reductions of erased expressions correspond directly to their erased counterpart. The proof of type safety of our system is built around this idea of correspondence, which is discussed in Section 4.4.

### *3.3. Unified Subtyping*

Figure 5 shows the (sub)typing rules of the system. We adopt a simplified design based on unified subtyping (Yang and Oliveira, 2017). The subtyping rules and typing rules are merged into a single judgment $\Gamma \vdash e_1 \leq e_2 : A$.

Unified subtyping solves the challenging issue of mutual dependency between typing and subtyping in a dependent type system. The interpretation of this judgment is "under context $\Gamma$, $e_1$ is a subtype of $e_2$ and they are both of type $A$". In this form of formalization, the typing judgment $\Gamma \vdash e : A$ is a special case of the unified subtyping judgment $\Gamma \vdash e \leq e : A$, and the well-formedness of types $\Gamma \vdash A$ is expressed by $\Gamma \vdash A : \star$.

*Subtyping Rules for Universal Quantifications.* The subtyping rules for universal quantifications (rules S-FORALL-L and S-FORALL-R) follow the spirit of the Odersky and Läufer's polymorphic subtyping (Odersky and Läufer, 1996; Dunfield and Krishnaswami, 2013), where the subtyping relation is interpreted as a "more-general-than" relation. A polymorphic type $\forall\, x : A.\ B$ is more general than another type $C$ when its well-typed instantiation is more general than $C$ (rule S-FORALL-L). A polymorphic type $\forall\, x : B.\ C$ is less general than a type $A$, if $C$ is less general than $A$ when the argument with the polytype $(x : B)$ is instantiated abstractly (rule S-FORALL-R).

$$
\begin{array}{llll}
\text{Erased Expressions} & E, A, B & ::= & x \mid n \mid k \mid \mathbb{N} \mid E_1\, E_2 \mid \lambda\, x.\ E \mid \Pi\, x : A.\ B \\
 & & \mid & \Lambda\, x.\ E \mid \forall\, x : A.\ B \mid \mu\, x.\ E \mid \mathrm{cast}_{\Uparrow}\, E \mid \mathrm{cast}_{\Downarrow}\, E \\
\text{Erased Values} & ev & ::= & k \mid n \mid \mathbb{N} \mid \lambda\, x.\ E \mid \Pi\, x : A.\ B \mid \Lambda\, x.\ E \\
 & & \mid & \forall\, x : A.\ B \mid \mathrm{cast}_{\Uparrow}\, E
\end{array}
$$

$$
\begin{array}{rrrr}
|x| = x & |n| = n & |k| = k & |\mathbb{N}| = \mathbb{N} \\
\multicolumn{2}{c}{|e_1\, e_2| = |e_1|\ |e_2|} & \multicolumn{2}{c}{|\mu\, x : A.\ e| = \mu\, x.\ |e|} \\
\multicolumn{2}{c}{|\lambda\, x : A.\ e| = \lambda\, x.\ |e|} & \multicolumn{2}{c}{|\Pi\, x : A.\ B| = \Pi\, x : |A|.\ |B|} \\
\multicolumn{2}{c}{|\Lambda\, x : A.\ e| = \Lambda\, x.\ |e|} & \multicolumn{2}{c}{|\forall\, x : A.\ B| = \forall\, x : |A|.\ |B|} \\
\multicolumn{2}{c}{|\mathrm{cast}_{\Uparrow}\, [A]\, e| = \mathrm{cast}_{\Uparrow}\, |e|} & \multicolumn{2}{c}{|\mathrm{cast}_{\Downarrow}\, e| = \mathrm{cast}_{\Downarrow}\, |e|}
\end{array}
$$

$\boxed{E_1 \Longrightarrow E_2}$ *(Erased Semantics)*

ER-APP
$$\frac{E_1 \Longrightarrow E_2}{E_1\, E_3 \Longrightarrow E_2\, E_3}$$

ER-BETA
$$\frac{}{(\lambda\, x.\ E_1)\, E_2 \Longrightarrow [E_2/x]\, E_1}$$

ER-ELIM
$$\frac{}{(\Lambda\, x.\ E_1)\, E_2 \Longrightarrow E_1\, E_2}$$

ER-MU
$$\frac{}{\mu\, x.\ E \Longrightarrow [\mu\, x.\ E/x]\, E}$$

ER-CASTDN
$$\frac{E_1 \Longrightarrow E_2}{\mathrm{cast}_{\Downarrow}\, E_1 \Longrightarrow \mathrm{cast}_{\Downarrow}\, E_2}$$

ER-CAST-INST
$$\frac{}{\mathrm{cast}_{\Downarrow}\, (\Lambda\, x.\ E) \Longrightarrow \mathrm{cast}_{\Downarrow}\, E}$$

ER-CAST-ELIM
$$\frac{}{\mathrm{cast}_{\Downarrow}\, (\mathrm{cast}_{\Uparrow}\, E) \Longrightarrow E}$$

Figure 4: Erased Expressions and Operational Semantics

$\boxed{\vdash \Gamma}$ *(Well-formed Context)*

WF-NIL
$$\frac{}{\vdash \varnothing}$$

WF-CONS
$$\frac{\vdash \Gamma \quad x \text{ fresh in } \Gamma \quad \Gamma \vdash A : k}{\vdash \Gamma,\, x : A}$$

$\boxed{\Gamma \vdash e_1 \le e_2 : A}$ *(Unified Subtyping)*

S-VAR
$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x \le x : A}$$

S-LIT
$$\frac{\vdash \Gamma}{\Gamma \vdash n \le n : \mathbb{N}}$$

S-INT
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} \le \mathbb{N} : \star}$$

S-STAR
$$\frac{\vdash \Gamma}{\Gamma \vdash \star \le \star : \Box}$$

ABS
$$\frac{\boxed{\Gamma \vdash A : k_1} \quad \Gamma,\, x : A \vdash B : k_2 \quad \Gamma,\, x : A \vdash e_1 \le e_2 : B}{\Gamma \vdash \lambda\, x : A.\ e_1 \le \lambda\, x : A.\ e_2 : \Pi\, x : A.\ B}$$

S-APP
$$\frac{\Gamma \vdash \tau : A \quad \Gamma \vdash e_1 \le e_2 : \Pi\, x : A.\ B}{\Gamma \vdash e_1\, \tau \le e_2\, \tau : [\tau/x]\, B}$$

S-PI
$$\frac{\Gamma \vdash A_2 \le A_1 : k_1 \quad \Gamma,\, x : A_1 \vdash B_1 : k_2 \quad \Gamma,\, x : A_2 \vdash B_1 \le B_2 : k_2}{\Gamma \vdash \Pi\, x : A_1.\ B_1 \le \Pi\, x : A_2.\ B_2 : k_2}$$

S-MU
$$\frac{\Gamma \vdash \tau : k \quad \Gamma,\, x : \tau \vdash \sigma : \tau}{\Gamma \vdash \mu\, x : \tau.\ \sigma \le \mu\, x : \tau.\ \sigma : \tau}$$

S-BIND
$$\frac{\boxed{\Gamma \vdash A : k} \quad \Gamma,\, x : A \vdash B : \star \quad \Gamma,\, x : A \vdash e_1 \le e_2 : B \quad x \notin \mathrm{FV}(|e_1|) \cup \mathrm{FV}(|e_2|)}{\Gamma \vdash \Lambda\, x : A.\ e_1 \le \Lambda\, x : A.\ e_2 : \forall\, x : A.\ B}$$

S-CASTUP
$$\frac{\Gamma \vdash A : k \quad A \longrightarrow B \quad \Gamma \vdash e_1 \le e_2 : B}{\Gamma \vdash \mathrm{cast}_{\Uparrow}\, [A]\, e_1 \le \mathrm{cast}_{\Uparrow}\, [A]\, e_2 : A}$$

S-CASTDN
$$\frac{\Gamma \vdash B : k \quad A \longrightarrow B \quad \Gamma \vdash e_1 \le e_2 : A}{\Gamma \vdash \mathrm{cast}_{\Downarrow}\, e_1 \le \mathrm{cast}_{\Downarrow}\, e_2 : B}$$

S-FORALL-L
$$\frac{\boxed{\Gamma \vdash A : k} \quad \Gamma \vdash \tau : A \quad \Gamma,\, x : A \vdash B : \star \quad \Gamma \vdash [\tau/x]\, B \le C : \star}{\Gamma \vdash \forall\, x : A.\ B \le C : \star}$$

S-FORALL-R
$$\frac{\boxed{\Gamma \vdash B : k} \quad \Gamma \vdash A : \star \quad \Gamma,\, x : B \vdash A \le C : \star}{\Gamma \vdash A \le \forall\, x : B.\ C : \star}$$

S-FORALL
$$\frac{\boxed{\Gamma \vdash A : k} \quad \Gamma,\, x : A \vdash B \le C : \star}{\Gamma \vdash \forall\, x : A.\ B \le \forall\, x : A.\ C : \star}$$

S-SUB
$$\frac{\Gamma \vdash e_1 \le e_2 : A \quad \Gamma \vdash A \le B : k}{\Gamma \vdash e_1 \le e_2 : B}$$

Syntactic Sugars

$$\Gamma \vdash e : A \triangleq \Gamma \vdash e \le e : A \qquad \Gamma \vdash A \triangleq \Gamma \vdash A \le A : \star \qquad \Gamma \vdash A \le B \triangleq \Gamma \vdash A \le B : \star$$

Figure 5: Unified Subtyping Rules of $\lambda_I^{\forall}$.

Notably, our formalization is not the direct generalization of Odersky and Läufer's polymorphic subtyping, as we mentioned in Section 2.2. As highlighted in Figure 5: we add rule S-FORALL that axiomatizes the subtyping relation between two universal types, and additional premises are added to rules S-FORALL-L and S-FORALL-R besides the ones that appear in rules $\leq\forall L''$ and $\leq\forall R''$ in Section 2.2. We discuss the motivations for these changes in more detail in Section 4.1.

*Mono-expression Restrictions.* As in other predicative relations (such as the one by Odersky and Läufer), the type arguments for instantiation in rule S-FORALL-L are required to be mono-expressions, which has cascading effects on typing rules of other expressions. The arguments for applications are required to be mono-expressions, and the whole fixpoint expression is required to be a mono-expression. We shall discuss the rationale to impose these restrictions in Section 4.

*Kind Restriction for Universal Types.* For the kinding of types, we mainly follow the design of the Calculus of Constructions (Coquand and Huet, 1988). However, we specifically restrict the polymorphic type $\forall\, x : A.\ B$ to only have kind $\star$, but not $\Box$. Without this restriction, types of kind $\star$ (such as $\mathbb{N}$) are able to have "polymorphic kinds" like $\forall\, x : \mathbb{N}.\ \star$ through rules S-FORALL-L, S-FORALL-R, and S-SUB, which significantly complicates the kinding reasoning in the metatheory. Practically speaking, "polymorphic kinds" are not very common, so this restriction has little impact on the expressiveness of our language.

Since we expect the types of well-typed expressions to be well-kinded, the restriction propagates to the introduction rule of $\forall$ types (rule S-BIND). In this rule, $B$ is required to only have kind $\star$ but not $\Box$. In contrast, for rule S-ABS, the type $B$ at a similar position can have any kind. As a result, implicit polymorphic functions at type level are *never well-typed*. For example, $\Lambda a : \star.\, \lambda b : n.\, a$ is not well-typed since it would have been a polymorphic function of type $\forall a : \star.\, a \to \star$, which is not well-kinded due to the kinding restriction of universal types. Moreover, type-level function applications that involve implicit abstractions ($\Lambda$ expressions) are also never well-typed because of the restriction. So well-typed non-deterministic implicit instantiation can never occur at the type-level.

*Computational Irrelevance of Implicit Parameters.* As mentioned in Section 2.2, our language does not handle computationally relevant implicit parameters. The direct operational semantics shown in Figure 3 chooses arbitrary mono-expressions to instantiate the implicit arguments, which potentially breaks type safety. Thus, we adopt a restriction in rule S-BIND that is similar to the Implicit Calculus of Constructions (ICC) (Miquel, 2001). We only allow the implicit parameters to occur in type annotations in the body of implicit abstraction, so that the choices of implicit parameters are not relevant at runtime. The type safety of the direct operational semantics is proved indirectly in Section 4.4 with the help of the erasure of expressions.

*Redundant Premises.* All the premises boxed by dash lines are redundant in a way that the system without them is proved equivalent to the system with them. These redundant premises are there to simplify the mechanized proofs of certain lemmas, but can be safely dropped in an actual implementation.

## 4. The Metatheory of $\lambda_I^{\forall}$

This section presents the metatheory of $\lambda_I^{\forall}$, and discusses several challenges that arose during the design of the typing rules to ensure desired subtyping and typing properties in our system. The three main results of the metatheory are: *transitivity of unified subtyping*, *type-soundness* and *completeness with respect to Odersky and Läufer's* polymorphic subtyping. Transitivity of subtyping is a general challenge for dependent type systems due to the mutual dependency of typing and subtyping, and the Odersky and Läufer style subtyping brings new issues to the table. For type-soundness, the key challenge is the non-deterministic and non-type-preserving nature of the reduction relation. To address this issue, we employ a type soundness proof technique that makes use of the erased reduction relation shown in Figure 4.

### *4.1. Polymorphic Subtyping in a Dependently Typed Setting*

The polymorphic subtyping relation by Odersky and Läufer features the following two rules:

$$\frac{\Gamma \vdash_{\text{DK}} \tau \qquad \Gamma \vdash_{\text{DK}} [\tau/x]\, A \leq B}{\Gamma \vdash_{\text{DK}} \forall x.\, A \leq B}\, \leq\forall L \qquad\qquad \frac{\Gamma,\, x \vdash_{\text{DK}} A \leq B}{\Gamma \vdash_{\text{DK}} A \leq \forall x.\, B}\, \leq\forall R$$

In order for the *well-formedness* (Dunfield and Krishnaswami, 2013) property (*If* $\Gamma \vdash_{DK} A \leq B$, *then* $\Gamma \vdash_{DK} A$ *and* $\Gamma \vdash_{DK} B$) to hold in Läufer and Odersky's system, these two rules rely on certain properties that do not hold in our dependently typed generalization. So we make several adjustments in our adaptation to address these issues, which result in the difference between our current system and a direct generalization mentioned in Section 2.2.

*Reverse Substitution of Well-Formedness.* Rule $\leq\forall L$ relies on the *reverse substitution* property, but this property does not hold in a dependently typed setting. Thus we need an alternative design that still ensures well-formedness, but without relying on the reverse substitution property.

The reverse substitution property is: *If* $\Gamma \vdash_{DK} [B/x]\, A$ *and* $\Gamma \vdash_{DK} B$, *then* $\Gamma,\, x \vdash_{DK} A$. That is if we have a type $A$ with all occurrences of $x$ substituted by $B$ and $B$ is well-formed, we can conclude that $A$ is well-formed under $\Gamma,\, x$. In a dependently typed setting, a possible form of generalization of this property would be: *If* $\Gamma \vdash [B/x]\, A : \star$ *and* $\Gamma \vdash B : C$, *then* $\Gamma,\, x : C \vdash A : \star$, which unfortunately does not hold. In a dependent type system, the values of expressions also matter during type checking besides their types, a counter-example of the

property is:

$$F : \mathbb{N} \to \star,\, a : F\ 42 \vdash (\lambda y : F\ \boxed{42}\ .\, \mathbb{N})\ a : \star$$
$$F : \mathbb{N} \to \star,\, a : F\ 42,\ \boxed{x : \mathbb{N}} \vdash (\lambda y : F\ \boxed{x}\ .\, \mathbb{N})\ a : \star$$

We cannot "reverse substitute" the 42 in the type annotation to a variable of the same type: the application expression depends specifically on the value 42 in order for the type of argument $a$ to match the type of the parameter. So we add a premise $\Gamma, x : A \vdash B : \star$ in rule S-FORALL-L to directly ensure the well-formedness of types in the conclusion.

*Strengthening of Contexts.* Rule $\leq\forall R$ relies on a strengthening lemma: *if* $\Gamma, x \vdash A$ *and* $x$ *does not occur in* $A$*, then* $\Gamma \vdash A$, which is trivial to prove in their system. However the admissibility of its generalization: *if* $\Gamma, x : B \vdash A : \star$ *and* $x$ *does not occur in* $A$*, then* $\Gamma \vdash A : \star$, is much more complicated to reason about. We can construct the following example:

$$F : \mathbb{N} \to \star,\, A : \star,\ \boxed{a : A} \vdash F\ ((\Lambda\, x : A.\ \lambda\, y : \mathbb{N}.\ y)\ 42) : \star$$

The variable $a$ does not appear in any expression, but plays a crucial role when considering the subtyping statement $\Gamma \vdash \forall x : A. \mathbb{N} \to \mathbb{N} \leq \mathbb{N} \to \mathbb{N}$, which arises when type-checking the application $(\Lambda\, x : A.\ \lambda\, y : \mathbb{N}.\ y)\ 42$. In this case, we cannot apply rule S-FORALL-L unless we find a well-typed instance for the polymorphic parameter. So the variable $a$ in the context is needed even though it does not occur anywhere in the final judgment. Note that, since our system has a fixpoint operator, theoretically we could construct a diverging program $\mu\, x : A.\ x$ to instantiate the implicit parameter, but this possibility leads to several other issues which we will discuss in Section 5.6. Furthermore, such an approach would not work for calculi without fixpoints.

Due to these complications, we assume that strengthening does not hold in our system for now. We add a premise $\Gamma \vdash A : \star$ to rule S-FORALL-R to work around this issue, requiring $A$ to be a well-kinded type without the help of the fresh variable. A consequence of adding this premise is that we will encounter a circular proof while trying to prove $\Gamma \vdash \forall\, x : A.\ B \leq \forall\, x : A.\ B : \star$, for arbitrary $A$ and $B$ by first applying rule S-FORALL-R. We resolve this issue by adding rule S-FORALL.

### *4.2. Typing Properties of* $\lambda_I^{\forall}$

With our rules properly set up, we can prove most of the basic properties using techniques borrowed from the *unifed subtyping* (Yang and Oliveira, 2017) approach. We introduce *reflexivity*, *weakening*, *context narrowing*, *substitution* and *type correctness* in this section.

**Theorem 1 (Reflexivity).** *If* $\Gamma \vdash e_1 \leq e_2 : A$*, then* $\Gamma \vdash e_1 : A$ *and* $\Gamma \vdash e_2 : A$.

Usually, a subtyping relation is reflexive when any well-formed type is a subtype of itself. With unified subtyping, the well-formedness of types is expressed

by subtyping relation as well, so the reflexivity looks more like the generalized *well-formedness* mentioned in the previous section. Reflexivity breaks down into two parts, *left reflexivity* and *right reflexivity*.

**Lemma 2 (Left Reflexivity).** *If* $\Gamma \vdash e_1 \leq e_2 : A$, *then* $\Gamma \vdash e_1 : A$.

**Lemma 3 (Right Reflexivity).** *If* $\Gamma \vdash e_1 \leq e_2 : A$, *then* $\Gamma \vdash e_2 : A$.

Both of the branches are proved by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : A$. Left reflexivity and right reflexivity when derivations end with rule S-FORALL-L and rule S-FORALL-R respectively are directly solved by rule S-FORALL.

**Theorem 4 (Weakening).** *If* $\Gamma_1, \Gamma_3 \vdash e_1 \leq e_2 : A$ *and* $\vdash \Gamma_1, \Gamma_2, \Gamma_3$, *then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \leq e_2 : A$.

*Weakening* is proved by induction on the derivation of $\Gamma_1, \Gamma_3 \vdash e_1 \leq e_2 : A$. The redundant premises discussed in Section 3.3 help to simplify the proof, by creating the induction hypotheses about the type annotation of various expressions. Otherwise, we are not able to prove $\vdash \Gamma_1, \Gamma_2, \Gamma_3, x : A$ with only $\vdash \Gamma_1, \Gamma_3, x : A$ given and no help from induction hypotheses.

**Theorem 5 (Context Narrowing).** *If* $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leq e_2 : C$ *and* $\Gamma_1 \vdash A \leq B : k$, *then* $\Gamma_1, x : A, \Gamma_2 \vdash e_1 \leq e_2 : C$.

**Lemma 6 (Well-formedness of Narrowing Context).** *If* $\vdash \Gamma_1, x : B, \Gamma_2$ *and* $\Gamma_1 \vdash A \leq B : k$, *then* $\vdash \Gamma_1, x : A, \Gamma_2$.

Theorem 5 and Lemma 6 are proved by mutual induction on the derivations of $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leq e_2 : C$ and $\vdash \Gamma_1, x : B, \Gamma_2$. Rule S-VAR is the only non-trivial case to solve: it relies on *weakening* to conclude $\Gamma_1, x : A, \Gamma_2 \vdash A \leq B : k$ from $\Gamma_1 \vdash A \leq B : k$, in order to derive $\Gamma_1, x : A, \Gamma_2 \vdash x : B$ through rule S-SUB.

**Theorem 7 (Substitution).** *If* $\Gamma_1, x : A, \Gamma_2 \vdash e_1 \leq e_2 : B$ *and* $\Gamma_1 \vdash \tau : A$, *then* $\Gamma_1, [\tau/x]\Gamma_2 \vdash [\tau/x]\, e_1 \leq [\tau/x]\, e_2 : [\tau/x]\, B$.

Notably *substitution* has a mono-expression restriction on the substituted expression. This is due to the mono-expression restriction on the instantiation of polymorphic parameters in rule S-FORALL-L.

Take the following derivation as an example:

$$\frac{A : \star,\ F : A \to \star,\ a : A \vdash [\,a\,/x]\, F\ x \leq F\ a : \star}{A : \star,\ F : A \to \star,\ a : A \vdash \forall x : A.\, F\ x \leq F\ a : \star}\ \text{S-FORALL-L}$$

Assuming that we have no mono-expression restrictions on *substitution* and rule S-APP. If we substitute $a$ with an arbitrary poly-expression, the derivation stops working because rule S-FORALL-L requires a mono-expression instantiation and rule S-APP requires the argument of both sides to be syntactically the same.

Worth mentioning is that while substitution of poly-expressions breaks the subtyping aspect of the language, a special case of the substitution theorem that discusses the typing of one expression (If $\Gamma_1, x : A, \Gamma_2 \vdash e : B$ and $\Gamma_1 \vdash e_1 : A$, then $\Gamma_1, [e_1/x]\,\Gamma_2 \vdash [e_1/x]\,e : [e_1/x]\,B$) does not hold for similar reasons. Because, in dependently typed languages, substitutions are also involved in the types of expressions as well. Due to the presence of rule S-SUB, we still have to maintain the potential subtyping relation of the types of expressions after substitution, for example:

$$\frac{A : \star,\ F : A \to \star,\ a : A,\ b : \forall x : A.\, F\ x \vdash b : \forall x : A.\, F\ x}{A : \star,\ F : A \to \star,\ a : A,\ b : \forall x : A.\, F\ x \vdash b : F\ a}\ \text{S-SUB}$$

As a result, the substitution theorem only holds with the mono-expression restriction. This has a cascading effect on typing rules like rules S-APP and S-MU whose expressions trigger substitutions during reduction. So the mono-expression restriction has to be added for those rules for the system to be type-safe.

**Lemma 8 (Context Well-formedness of Substitution).** *If* $\vdash \Gamma_1, x : A, \Gamma_2$ *and* $\Gamma_1 \vdash \tau : A$, *then* $\vdash \Gamma_1, [\tau/x]\,\Gamma_2$.

After understanding the mono-expression restriction on *substitution*, the actual proof is not complicated: it proceeds by mutual induction with Lemma 8 on the derivations of $\Gamma_1, x : A, \Gamma_2 \vdash e_1 \leq e_2 : B$ and $\vdash \Gamma_1, x : A, \Gamma_2$. When the derivation ends with rules S-CASTUP and S-CASTDN, the proof requires the reduction relation to preserve after the substitution. This property should usually hold, but it puts an interesting constraint which we have to consider when designing the reduction rules (see Section 5.2).

**Lemma 9 (Reduction Substitution).** *If* $A \longrightarrow B$, *then* $[\tau/x]\,A \longrightarrow [\tau/x]\,B$

**Theorem 10 (Type Correctness).** *If* $\Gamma \vdash e_1 \leq e_2 : A$, *then* $\exists k.\, \Gamma \vdash A : k$ *or* $A = \Box$.

*Type correctness* is a nice property that ensures that what appears in the position of a type is actually a type. The theorem is proved by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : A$. The only non-trivial case is when the derivation ends with rule S-APP. We make use of the substitution lemma and the inductive hypothesis to demonstrate the head of a $\Pi$ type preserves its kind after the argument is applied.

### *4.3. Transitivity*

Transitivity is typically one of the most challenging properties to prove in calculi with subtyping and it was also one of the harder proofs in $\lambda_I^{\forall}$. The proof of transitivity requires a generalization of the usual transitivity property:

**Theorem 11 (Generalized Transitivity).** *If* $\Gamma \vdash e_1 \leq e_2 : A$ *and* $\Gamma \vdash e_2 \leq e_3 : B$*, then* $\Gamma \vdash e_1 \leq e_3 : A$*.*

where the types of the premises are potentially different. To prove this property we employ sizes for the inductive argument. Moreover we rely on a subtle property of uniqueness of kinds.

*Uniqueness of Kinds.* Assuming that the second premise of generalized transitivity is derived by rule S-FORALL-R, then we face the following problem:

$$\frac{\Gamma \vdash e_1 \leq e_2 : A \qquad \Gamma, x : B \vdash e_2 \leq C : \star}{\Gamma \vdash e_1 \leq \forall\, x : B.\ C : A}$$

Before applying rule S-FORALL-R to the conclusion, we have to establish the relationship between $A$ and $\star$. Were there no restrictions on the kinding of $\forall$ types, this would be a much more complicated situation, where the inversion lemmas of about kinds and transitivity depend on each other. This is one of the main reasons why we forbid $\forall$ types having kind $\Box$. Then we can have the following theorem:

**Theorem 12 (Kind Uniqueness).** *If* $\Gamma \vdash e : k$ *and* $\Gamma \vdash e : A$*, then* $A = k$*.*

The proof is achieved by generalizing the shape of $k$ to be $\Pi x : A. \ldots \Pi x : B. \ldots k$ for obtaining useful inductive hypotheses when $\Gamma \vdash e : k$ is derived by rule S-APP. Then the proof proceeds with induction on the derivation of the generalized $\Gamma \vdash e : k$ and assembling various lemmas of kinding to solve different cases.

With the help of *kind uniqueness*, we ensure the equivalence of $A$ and $\star$ on this and other similar situations.

*The Induction.* We prove *generalized transitivity* by performing a strong induction on the ordered 3-tuple of measures:

$$\langle \#\forall(e_1) + \#\forall(e_2) + \#\forall(e_3),\ \mathbf{size}(e_1) + \mathbf{size}(e_3),\ \mathcal{D}_1 + \mathcal{D}_2 \rangle$$

where

- $\#\forall(e)$ counts the number of $\forall$ quantifiers in expression $e$, which solves cases when either side of the premise is derived by rules S-FORALL, S-FORALL-L, and S-FORALL-R.
- $\mathbf{size}(e)$ measures the size of the syntax tree of expression $e$. The sum of expression sizes solves most of the other standard recursive cases.
- $\mathcal{D}_1$ and $\mathcal{D}_2$ denote the sizes of the derivation trees of the first and the second premise respectively. The sum of sizes of derivation tree solves the case involving rule S-SUB where the sizes of expressions do not decrease.

The proof is mainly inspired by DK's transitivity proof of their declarative subtyping system of induction on the pair of $\langle \#\forall(e_2),\ \mathcal{D}_1 + \mathcal{D}_2 \rangle$ (Dunfield and Krishnaswami, 2013), with some adjustments to fit in our system.

The most problematic case to solve is when the first premise is derived by rule S-FORALL-R, and the second is derived by rule S-FORALL-L. Essentially we have to show the following:

$$\frac{\Gamma,\, x : A \vdash e_1 \leq B : \star \qquad \Gamma \vdash [\tau/x]\, B \leq e_3 : \star \qquad \Gamma \vdash \tau : A}{\Gamma \vdash e_1 \leq e_3 : \star}$$

Here the only decreasing measure we can rely on is that $\#\forall([\tau/x]\, B)$ is one less than $\#\forall(\forall\, x : A.\ B)$ (since $\tau$ is a monotype which does not contain any $\forall$ quantifier). To solve this case, we first perform a substitution on the premise $\Gamma,\, x : A \vdash e_1 \leq B : \star$ with the help of the fact that $x$ does not occur in $e_1$, obtaining $\Gamma \vdash e_1 \leq [\tau/x]\, B : \star$, then we use the inductive hypothesis provided by $\#\forall(e_2)$.

The reason why we cannot directly copy DK's proof measure is because of the case when both premises end with rule S-PI, where we encounter the following problem:

$$\frac{\Gamma,\, x : A_2 \vdash B_1 \leq B_2 : k \qquad \Gamma,\, x : A_3 \vdash B_2 \leq B_3 : k \qquad \Gamma \vdash A_3 \leq A_2 : k_2}{\Gamma,\, x : A_3 \vdash B_1 \leq B_3 : k}$$

The first and the second premise above do not share the same context, which must be unified with the context narrowing theorem to be able to use the inductive hypothesis. However context narrowing potentially increases the size of the derivation tree, so we are not able to use the inductive hypothesis of $\mathcal{D}_1 + \mathcal{D}_2$, and resort to the sizes of expressions $(\mathbf{size}(e_1) + \mathbf{size}(e_3))$

We have to make adjustments to solve the cases which preserve the size of derivation tree, but not the sizes of the expressions, which is when the first premise is derived by rule S-FORALL-L:

$$\frac{\Gamma \vdash [\tau/x]\, B \leq e_2 : \star \qquad \Gamma \vdash e_2 \leq e_3 : C}{\Gamma \vdash \forall\, x : A.\ B \leq e_3 : \star}$$

In this case, $\#\forall([\tau/x]\, B)$ is one less than $\#\forall(\forall\, x : A.\ B)$, so it can be solved by applying rule S-FORALL-L and the inductive hypothesis of $\#\forall(e_1)$. Additionally, $\#\forall(e_3)$ is added to make the measure "symmetric" to handle the contravariance case of rule S-PI.

Then, most of the cases that do not involve $\forall$ can be solved by applying the inductive hypothesis corresponding to $\mathbf{size}(e_1)+\mathbf{size}(e_3)$. Finally, $\mathcal{D}_1+\mathcal{D}_2$ solves the cases where either premise ends with rule R-SUB, where the only decreasing measure is the size of the derivation trees when the sizes of expressions remain the same.

**Corollary 13 (Transitivity).** *If* $\Gamma \vdash e_1 \leq e_2 : A$ *and* $\Gamma \vdash e_2 \leq e_3 : A$*, then* $\Gamma \vdash e_1 \leq e_3 : A$*.*

*Transitivity* is a special case of *generalized transitivity* where $A = B$.

*4.4. Type Safety*

Since the reduction rules of $\lambda_I^\forall$ do not have access to typing information, they cannot perform valid instantiation checks of the implicit arguments during applications. Thus, the runtime semantics is non-deterministic and potentially non-type-preserving. We tackle this issue by employing designs that make the choices of implicit instantiations irrelevant at runtime with the occurrence restrictions in rule S-BIND. We define an erasure function (shown in Figure 4) that eliminates the type annotations in some expressions ($\lambda$, $\Lambda$, $\mu$ and $\text{cast}_{\Uparrow}$), and keep implicit parameters from occurring in the erased expressions. This way the choices of implicit instantiations only affect type annotations, which are not relevant for runtime computation.

We show that $\lambda_I^\forall$ is type-safe in the sense that, if an expression is well-typed, then the reduction of its erased version does not "go wrong". Figure 4 shows the semantics of erased expressions. The erasure semantics mostly mirrors the semantics shown in Figure 3, except for rules ER-ELIM and ER-CAST-ELIM, which conveys the idea of the irrelevance of implicit instantiation by eliminating the parameter directly.

*Progress.* We show the *progress* property for both the original expressions and the erased expressions.

**Theorem 14 (Generalized progress).** *If* $\varnothing \vdash e_1 \leq e_2 : A$*, then* $\exists\, e_1'.\, e_1 \longrightarrow e_1'$ *or* $e_1$ *is a value, and* $\exists\, e_2'.\, e_2 \longrightarrow e_2'$ *or* $e_2$ *is a value.*

**Theorem 15 (Generalized progress on erased expressions).** *If* $\varnothing \vdash e_1 \leq e_2 : A$*, then* $\exists\, E_1'.\, |e_1| \Longrightarrow E_1'$ *or* $|e_1|$ *is an erased value, and* $\exists\, E_2'.\, |e_2| \Longrightarrow E_2'$ *or* $|e_2|$ *is an erased value.*

We prove a generalized version of *progress* that involves both sides of the expressions with unified subtyping. Note that they are not necessarily simultaneously reducible or irreducible due to the presence of rules S-FORALL-L and S-FORALL-R. The left-hand-side may be reducible with the right-hand-side being a value or vice versa.

Both theorems are proved by induction on the derivation of $\varnothing \vdash e_1 \leq e_2 : A$. The proof is mostly straightforward except when the derivation ends with $\text{cast}_{\Downarrow}$, where we have to show that the inner expressions $e$ of $\text{cast}_{\Downarrow}\ e$ either reduces, or is a $\text{cast}_{\Uparrow}$ or a $\Lambda$-expression. We prove another fact to solve the situation: for a well-typed expression whose type reduces, that expression cannot be a value unless it is a $\text{cast}_{\Uparrow}$ or $\Lambda$-expression.

**Lemma 16 (Reducible Type).** *If* $\Gamma \vdash e : A$*,* $A \longrightarrow B$ *and* $e$ *is not* $\text{cast}_{\Uparrow}$ *or a* $\Lambda$*-expression, then* $e$ *is not a value.*

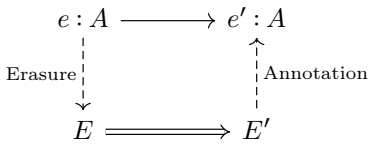


Figure 6: Diagram for Erased Preservation without Subtyping

**Lemma 17 (Erased Value to Value).** *If $|e|$ is an erased value, then $e$ is a value.*

This lemma is also useful for the proof of progress for erased expressions. Since the value definitions are very similar, we can use the property of values on erased values.

**Lemma 18 (Value to Erased Value).** *If $e$ is a value, then $|e|$ is an erased value.*

**Corollary 19 (Progress).** *If $\varnothing \vdash e : A$, then $\exists\, e'.\, e \longrightarrow e'$ or $e$ is a value.*

**Corollary 20 (Progress on erased expressions).** *If $\varnothing \vdash e : A$, then $\exists\, E'.\, |e| \Longrightarrow E'$ or $|e|$ is an erased value.*

Both corollaries directly follow from their generalized versions.

*Preservation.* The direct operational semantics is not generally type-preserving and deterministic because of the implicit instantiations. Thus, we show preservation with the help of the erased expressions (where implicit parameters do not matter to the computation). For other reduction rules that do not involve such issues, we discuss them as though we are proving a normal preservation for brevity.

**Theorem 21 (Subtype Preservation).** *If $\Gamma \vdash e_1 \leq e_2 : A$, $|e_1| \Longrightarrow E_1'$ and $|e_2| \Longrightarrow E_2'$, then $\exists\, e_1'\, e_2'.\ |e_1'| = E_1'$, $|e_2'| = E_2'$, $e_1 \longrightarrow e_1'$, $e_2 \longrightarrow e_2'$ and $\Gamma \vdash e_1' \leq e_2' : A$.*

The theorem might look a little complicated at first glance. It breaks down into two aspects: the erasure-annotation process and the subtype preservation.

Figure 6 shows the idea of our preservation lemma without considering the subtyping aspect (assuming $\Gamma \vdash e : A$ instead of $\Gamma \vdash e_1 \leq e_2 : A$). Here we use *annotation* as the reverse process of erasure. If an expression ($e$) is well-typed, and its erasure ($E$) reduces to another erased expression ($E'$), we can find a "annotated" expression of $E'$ ($e'$) that is reduced by $e$ and also preserves the type $A$. When no implicit instantiation happens in the reduction, then $e \longrightarrow e'$ is deterministic: i.e. it is just normal type preservation. When there are implicit instantiations, if the erased expression can reduce, we show that there exists a valid instantiation for $e$ that preserves its type after the reduction, and

this instantiation only affects type annotations. In other words, the runtime semantics of $\lambda_I^{\forall}$ can be implemented only with erased expressions.

Aside from the erasure aspect of our preservation lemma, we also consider the generalized version of preservation in our unified subtyping system, the *subtype preservation*, where reductions not only preserve the type of expressions, they also preserve the subtyping relation between expressions as well.

The theorem is proved by induction on the derivation of $\Gamma \vdash e_1 \leq e_2 : A$, cases for rules R-BETA and R-MU are solved with the substitution theorem, cases rules R-APP and R-CASTDN are solved by inductive hypotheses. The interesting cases to prove are rule R-CAST-ELIM and cases involving implicit instantiation (rules R-INST and R-CAST-INST).

*Cast Elimination.* The main issue of the cast elimination case can be demonstrated by the following derivation:

$$\dfrac{\dfrac{B_1 \longrightarrow B_2 \qquad \dfrac{\Gamma \vdash A_1 \leq B_1 : k \qquad \dfrac{A_1 \longrightarrow A_2 \qquad \Gamma \vdash e : A_2}{\Gamma \vdash \mathrm{cast}_{\Uparrow}\,[A_1]\,e : A_1}\ \text{S-CASTUP}}{\Gamma \vdash \mathrm{cast}_{\Uparrow}\,[A_1]\,e : B_1}\ \text{S-SUB}}{\Gamma \vdash \mathrm{cast}_{\Downarrow}\,(\mathrm{cast}_{\Uparrow}\,[A_1]\,e) : B_2}\ \text{S-CASTDN}$$

Here the typing of the inner $\mathrm{cast}_{\Uparrow}$ is not directly derived by rule R-CASTUP, but by the subsumption rule instead. We want to show that after the cast elimination (following rule R-CAST-ELIM), expression $e$ has type $B_2$, while in reality it has type $A_2$ (as highlighted). Therefore want to show $\Gamma \vdash A_2 \leq B_2$ with the information that $\Gamma \vdash A_1 \leq B_1$, $A_1 \longrightarrow A_2$ and $B_1 \longrightarrow B_2$, which depends on the property we want to prove initially, subtype preservation. Since subtype preservation needs to solve the cast elimination case, here we have a circular dependency of properties. This problem was also observed by Yang and Oliveira (2017). They solved this situation by a delicate approach with the help of an essential lemma *reduction exists in the middle* (If $\Gamma \vdash e_1 \leq e_2 : A$, $\Gamma \vdash e_2 \leq e_3 : A$ and $e_1 \longrightarrow e_1'$, $e_3 \longrightarrow e_3'$, there exists $e_2'$ such that $e_2 \longrightarrow e_2'$). Unfortunately this lemma does not hold in our system since universal types, which are not reducible, can appear in the middle of two reducible types, so we cannot adopt their proof on this case.

We tackle this problem from another direction, with the observation that the demand for subtype preservation property shifts from the term-level to the type-level. With the Calculus-of-Constructions-like kind hierarchy, our system only has limited layers in types (type $\mathbb{N}$ has kind $\star$, kind $\star$ has kind $\Box$). In fact, we only need to go up one layer in the type hierarchy to be able to obtain subtype preservation directly, since there is no subtyping at the kind level, hence no problem for the cast elimination there. Even better, we show that by going up one level in the type hierarchy (only discussing the types of terms), the options for the reduction that can be performed by a well-typed term are very limited. Implicit abstractions cannot occur in type computation due to the kind

$\boxed{A_1 \longrightarrow_D A_2}$ *(Deterministic Reduction)*

$$\text{DR-APP}\ \frac{e_1 \longrightarrow_D e_2}{e_1\ e_3 \longrightarrow_D e_2\ e_3} \qquad \text{DR-BETA}\ \frac{}{(\lambda\, x : A.\ e_1)\ e_2 \longrightarrow_D [e_2/x]\, e_1}$$

$$\text{DR-MU}\ \frac{}{\mu\, x : A.\ e \longrightarrow_D [\mu\, x : A.\ e/x]\, e}$$

Figure 7: Deterministic Reduction.

restriction of universal types as explained in Section 3.3. Furthermore, we also prove that well-typed reductions never occur for kinds, so cast operators also do not occur in type-level computation.

Figure 7 shows the effective reduction rules inside cast operators.

**Lemma 22 (Deterministic Reduction).** *If* $A \longrightarrow_D A_1$ *and* $A \longrightarrow_D A_2$, *then* $A_1 = A_2$.

**Lemma 23 (Deterministic Type Reduction).** *If* $\Gamma \vdash A_1 : k$ *and* $A_1 \longrightarrow A_2$, *then* $A_1 \longrightarrow_D A_2$.

The cases for implicit abstractions are easy to prove. For the cast operators we have the following lemma.

**Lemma 24 (Expressions of kind $\Box$ are never reduced).** *If* $A \longrightarrow B$ *and* $\Gamma \vdash e : A$, *then* $B$ *does not have kind* $\Box$.

At first sight, the result of this lemma may be surprising because it means that we cannot construct a reducible expression like: $(\lambda\, x : \mathbb{N}.\ \star)\ 42$ which has kind $\Box$. In reality, the lambda expression must be of type $\mathbb{N} \to \Box$ for the application to be well-typed. However, as we employ the conventional typing rule for lambda abstractions of Calculus of Constructions (Coquand and Huet, 1988), the function types of the lambda abstractions themselves must be well-kinded (see rule S-ABS). Since $\Box$ itself does not have a kind, $\mathbb{N} \to \Box$ is not well-kinded, therefore the whole application is not well-typed. For this reason, the position where kind $\star$ can occur in a well-typed expression is very restricted, hence the lemma is provable.

With the previous lemmas, the subtype preservation lemma for type computation is easily shown.

**Lemma 25 (Subtype Preservation for Types).** *If* $\Gamma \vdash A_1 \leq B_1 : k$, $A_1 \longrightarrow_D A_2$ *and* $B_1 \longrightarrow_D B_2$, *then* $\Gamma \vdash A_2 \leq B_2 : k$.

*Implicit Instantiations.* The proof of two cases for implicit instantiations (rules R-INST and R-CAST-INST) are quite similar. In our language, implicit instantiations of implicit functions are only triggered by rule S-FORALL-L, which is exactly where polymorphic types are instantiated. The implicit argument is the same mono-expression ($\tau$) that instantiates the polymorphic type in rule S-FORALL-L. The type of the instantiation result of $\Lambda$ expression is the same as the instantiation of the polymorphic types, with the same argument.

With the observations above, the remaining proofs are finished by standard inversion lemmas, with the help of the *substitution* theorem to handle type preservation after the instantiations.

### *4.5. Equivalence to a Simplified System*

We mentioned in Section 3.3 that the premises boxed by dashed lines in the unified subtyping rules are redundant. They help in the formalization, but the calculus is equivalent to a variant of the calculus without them. We define unified subtyping relation $\Gamma \vdash_s e_1 \leq e_2 : A$, whose rules are the same as the unified subtyping rules of $\lambda_I^\forall$, but with all redundant premises eliminated. Also, rules S-CASTDN and S-CASTUP are simplified to use deterministic reduction ($A \longrightarrow_D B$) instead of the reduction rule $A \longrightarrow B$ as shown below (other rules are omitted):

$\boxed{\Gamma \vdash_s e_1 \leq e_2 : A}$ *(Simplified Unified Subtyping)*

SS-CASTDN

$$\frac{\Gamma \vdash_s e_1 \leq e_2 : A \qquad A \longrightarrow_D B \qquad \Gamma \vdash_s B : k}{\Gamma \vdash_s \mathrm{cast}_{\Downarrow} e_1 \leq \mathrm{cast}_{\Downarrow} e_2 : B}$$

SS-CASTUP

$$\frac{\Gamma \vdash_s e_1 \leq e_2 : B \qquad A \longrightarrow_D B \qquad \Gamma \vdash_s A : k}{\Gamma \vdash_s \mathrm{cast}_{\Uparrow} [A] e_1 \leq \mathrm{cast}_{\Uparrow} [A] e_2 : B}$$

We prove that the two system are equivalent in terms of expressiveness.

**Theorem 26 (Equivalence of $\lambda_I^\forall$ and the Simplification).** *If $\Gamma \vdash e_1 \leq e_2 : A$ then $\Gamma \vdash_s e_1 \leq e_2 : A$. And if $\Gamma \vdash_s e_1 \leq e_2 : A$ then $\Gamma \vdash e_1 \leq e_2 : A$.*

### *4.6. Subsumption of Polymorphic Subtyping*

Finally we show that the subtyping aspect of $\lambda_I^\forall$ subsumes Odersky and Läufer's polymorphic subtyping (Odersky and Läufer, 1996). To be more precise we show that our unified subtyping relation subsumes DK's declarative subtyping relation (Dunfield and Krishnaswami, 2013), whose syntax and subtyping relation are shown in Figure 1.

Figure 8 shows the transformation from DK's types to $\lambda_I^\forall$'s types. Then we prove the subsumption in terms of type well-formedness and subtyping by following the interpretation of unified subtyping.

**Theorem 27 (Subsumption of Type Well-formedness).** *If $\Gamma \vdash_{DK} A$, then $\lceil \Gamma \rceil \vdash \lceil A \rceil : \star$*

$$\lceil x \rceil = x \qquad \lceil \mathbb{N} \rceil = \mathbb{N} \qquad \lceil A \to B \rceil = \lceil A \rceil \to \lceil B \rceil \qquad \lceil \forall x.\, A \rceil = \forall x : \star.\, \lceil A \rceil$$

$$\lceil \varnothing \rceil = \varnothing \qquad\qquad \lceil \Gamma,\, x \rceil = \lceil \Gamma \rceil,\, x : \star$$

Figure 8: Lifting Types and Contexts in Polymorphic Subtyping to $\lambda_I^\forall$

Straightforward. For the case where $A = \forall x.B$, rule S-FORALL can be used directly bypassing the complications of rule S-FORALL-L and rule S-FORALL-R.

**Theorem 28 (Subsumption of Polymorphic Subtyping).** *If* $\Gamma \vdash_{DK} A \leq B$*, then* $\lceil \Gamma \rceil \vdash \lceil A \rceil \leq \lceil B \rceil : \star$

The interesting cases are when the premise is derived by $\leq\forall L$ or $\leq\forall R$, because of the addition of premises in our generalized system ($\Gamma,\, x : A \vdash B : \star$ in rule S-FORALL-L, $\Gamma \vdash A : \star$ in rule S-FORALL-R). Both cases can be solved with the help of the *well-formedness* lemma in DK's system. We can conclude $\Gamma, x \vdash_{\mathrm{DK}} A$ from $\Gamma \vdash_{\mathrm{DK}} \forall x.\, A \leq B$ for the $\leq\forall L$ case, and conclude $\Gamma \vdash_{\mathrm{DK}} A$ from $\Gamma \vdash_{\mathrm{DK}} A \leq \forall x.\, B$ for the $\leq\forall R$ case. Then Theorem 27 can be used to solve the additional premises in $\lambda_I^\forall$.

## 5. Discussions and Future Work

In this section we discuss some design choices and alternatives, as well as possible future work.

### *5.1. Open Terms Reduction*

Usually, only closed terms are considered when designing the operational semantics, and irreducible open terms like variables are not considered to be values. However, since in $\lambda_I^\forall$ the reduction relation $e_1 \longrightarrow e_2$ is also used in the unified subtyping relation, the reduced expressions can be open terms and well-typed under contexts. Yang and Oliveira (2017) observed this issue and included *inert terms* (Accattoli and Guerrieri, 2016) as values to handle the situation that open terms like $x\ e$ (applying variable $x$ to the argument $e$) is also irreducible, which considerably complicates their metatheory around the operational semantics. Fortunately, our call-by-name reduction rules do not rely on the notion of value, which for closed terms, represents irreducible forms. In other words, there is no premise in our reduction rules that require some expression to be a value or irreducible. Thus, the definition of values or irreducible terms does not matter when reasoning about type-level call-by-name reductions, and we do not have to complicate the metatheory of our system with inert terms.

*5.2. Call-by-value Semantics of Cast Operators*

An alternative design around cast operators is the call-by-value (CBV) style (Yang and Oliveira, 2019), by not considering all $\text{cast}_{\Uparrow}$ terms as values, and performing cast elimination only when the expression inside two casts is a value. Such design requires us to have a more general definition of value, and there would be a need for inert terms as mentioned in Section 5.1

However, a simple design with CBV-style cast semantics and inert terms potentially leads to a system where *reduction substitution* (see Section 4.2) does not hold. With the CBV-style reduction rules, which consist of the following rules:

$$\textsc{cbv-r-castdn}\ \frac{e_1 \longrightarrow_{\text{cbv}} e_2}{\text{cast}_{\Downarrow}\, e_1 \longrightarrow_{\text{cbv}} \text{cast}_{\Downarrow}\, e_2} \qquad \textsc{cbv-r-castup}\ \frac{e_1 \longrightarrow_{\text{cbv}} e_2}{\text{cast}_{\Uparrow}\, [A]\, e_1 \longrightarrow_{\text{cbv}} \text{cast}_{\Uparrow}\, [A]\, e_2}$$

$$\textsc{cbv-r-cast-elim}\ \frac{\text{value}\ e}{\text{cast}_{\Downarrow}\, (\text{cast}_{\Uparrow}\, [A]\, e) \longrightarrow_{\text{cbv}} e}$$

Assuming that the notion of value is properly defined to capture irreducible open terms, these rules allow reductions like:

$$\text{cast}_{\Downarrow}\, (\text{cast}_{\Uparrow}\, [A]\, (f\ x)) \longrightarrow_{\text{cbv}} f\ x$$

The *reduction substitution* property breaks if we substitute $f$ to a lambda expression:

$$[\lambda y : B.\, x/f]\, \text{cast}_{\Downarrow}\, (\text{cast}_{\Uparrow}\, [A]\, (f\ x)) \longrightarrow_{\text{cbv}} \text{cast}_{\Downarrow}\, (\text{cast}_{\Uparrow}\, [A]\, x)$$

The reduction rule prioritizes reducing the inner expression of two casts, while *reduction substitution* expects $[\lambda y : B.\, x/f]\, \text{cast}_{\Downarrow}\, (\text{cast}_{\Uparrow}\, [A]\, (f\ x))$ to reduce to $(\lambda y : B.\, y)\ x$.

So we stick with the call-by-name style semantics for cast operators for now and leave the discussion of other possibilities of design in future work.

*5.3. Kind Restrictions on Polymorphic Types*

Currently, we impose restrictions on the kinding of polymorphic types ($\forall x : A.\, B$) to require that they only have kind $\star$ but not $\Box$. We believe that this has little impact on the usability of our system since polymorphic kinds such as $\forall\, x : \mathbb{N}.\ \star$ do not appear frequently in practice. It would be reasonable not to have this restriction, but this would complicate the development of the metatheory significantly.

One of the obstacles to removing the kind restriction is that there is a mutual dependency between the transitivity theorem and the subtyping reasoning of

polymorphic kinds. We wish to have some lemma like this:

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : \star}{\Gamma \vdash \star \leq A : \Box}$$

which depends on transitivity when the derivation of $\Gamma \vdash e : A$ ends with subsumption rule (rule S-SUB).

Note that the reverse variant of the lemma (If $\Gamma \vdash e : A$ and $\Gamma \vdash e : \star$ then $\Gamma \vdash A \leq \star : \Box$) is not generally true. A counter example is $A : \star \vdash \forall a : A. \star \leq \star$, which does not hold if we are unable to find a well-typed instantiation of an arbitrary type $A$. In turn, this breaks transitivity, when the derivation of the first premise ends with rule S-FORALL-L:

$$\frac{\Gamma \vdash [\tau/x]\, e_1 \leq e_2 : \star \qquad \Gamma \vdash e_2 \leq e_3 : B \qquad \Gamma \vdash \tau : A}{\Gamma \vdash \forall\, x : A.\ e_1 \leq e_3 : \star}$$

We cannot apply rule S-FORALL-L unless we can show $e_3$ is of type $\star$. One of the possibilities in this scenario is to show $\Gamma \vdash B \leq \star : \Box$, which does not always hold for the reason discussed above.

Moreover, we expect complexities while reasoning about the kinding of types after we lose the kinding uniqueness in other parts of the metatheory. Therefore we leave the relaxation of the kinding restrictions for future work.

### *5.4. Runtime Relevance of Implicit Arguments*

In our language, the implicit arguments have no computational impact at runtime and only provide the necessary scoping for type annotations. This is a similar design to ICC (Miquel, 2001) and ICC* (Barras and Bernardo, 2008) to simplify the development of a direct operational semantics for our language. Such a restriction can be lifted if we prove the runtime type-safety by elaboration to a second language, such as *the Calculus of Constructions* (Coquand and Huet, 1988), instead of providing the direct operational semantics.

We can elaborate implicit function types (universal types) to $\Pi$ types, implicit abstractions to lambda expressions, and implicit instantiations to explicit applications during type checking (when we have full information about the choice of implicit instantiations). However the elaboration on a unified subtyping system is not an easy task. The subtyping relation cannot simply be interpreted as a coercion between values of different types, since some of our subtyping rules involve a relation between terms instead of types. In other words, unified subtyping generalizes conventional subtyping relations that are defined on types only, to a relation defined on general terms. Therefore we leave the exploration of how elaboration can be done on a unified subtyping system as future work.

### *5.5. Algorithmic System and Challenges*

$\lambda_I^{\forall}$ does not currently have an algorithmic system since we consider a formalized algorithm for dependent type system itself a substantial challenge. Thus, an

algorithmic formulation it is left for future work. While comparing to existing algorithmic systems for higher-ranked polymorphic type inference for System F-like languages (Dunfield and Krishnaswami, 2013; Zhao et al., 2019), we identify one of the interesting challenges to develop an algorithmic system for $\lambda_I^{\forall}$.

In dependent type systems, the type of applications potentially depends on the values of their arguments. Therefore the unification problem we meet is potentially inside binders and depends on the value of the arguments. For an example in $\lambda_I^{\forall}$, consider:

$$\lambda F : \mathbb{N} \to \star.\, \lambda f : F\ 42 \to \mathbb{N}.\, f((\Lambda A : \star.\, \mu x : A.\, x)\ 42)$$

Here the type of $\Lambda A : \star.\, \mu x : A.\, x$ should be $\forall A : \star.\, A$. In a non-dependent setting, one can easily conclude the instantiation for type $A$ to be $F\ 42 \to \mathbb{N}$. However in our system, the type of $A$ is a dependent function type, with the shape of $(\Pi x : \mathbb{N}.\, F\ e)$, where expression $e$ satisfies the equation $[42/x]\, e = 42$ according to rule S-APP. Here we have two choices for the instantiation of $e$, namely $x$ and 42, but neither choice ($\Pi x.\, F\ x$ or $\Pi x.\, F\ 42$) for type $A$ leads to a more general solution than choosing the other.

Notably, for a similar reason, the decidable pattern fragment of higher-order unification (Miller, 1991a) specifically forbids scenarios where a unification variable applies to constants. The case shown above is similar, where the "application" is $A\ 42$ (dependent function type $A$ "applies" to 42) with unification variable $A$. Nonetheless, because our system does not allow implicit type-level computation (hence unifications remain at first-order), the choices of $A$ are restricted to $\Pi x : \mathbb{N}.\, F\ e_1$ where $[42/x]\, e_1 = 42$. With higher-order unification, we would instead have $\Pi x : \mathbb{N}.\, e_2$ where $[42/x]\, e_2 = F\ 42$.

There are potentially multiple approaches to this problem. For example, we can impose a similar restriction to the pattern unification and refuse to solve this kind of conflict entirely. Alternatively, we can only infer non-dependent function types when facing a unification variable, which is the choice by Eisenberg (2016). But which is the better method for our system remains to be studied.

### *5.6. The Issues of Strengthening*

The current lack of a proof for *strengthening* leads to non-trivial changes during the generalization of polymorphic subtyping relation as described in Section 4.1. It would be nice to be able to have strengthening to simplify $\lambda_I^{\forall}$. However the issues are quite tricky.

*Type Inhabitation.* As we mentioned earlier, one of the issues is centered around not being able to find a well-typed instantiation for an arbitrary type in general. Some instantiations can only be found in the presence of seemingly unused variables, which is a key reason for strengthening not holding in general. For example:

$$A : \star,\ a : A \vdash (\Lambda\, x : A.\ \lambda\, y : \mathbb{N}.\ y)\ 42 : \mathbb{N}$$

Here, to conclude that $\Lambda\, x : A.\ \lambda\, y : \mathbb{N}.\ y$ has type $\mathbb{N} \to \mathbb{N}$, we want to derive $\forall x : A. \mathbb{N} \to \mathbb{N} \leq \mathbb{N} \to \mathbb{N}$ by using rule S-FORALL-L. That would require a well-typed instantiation for the abstract type $A$, which is where the variable $a$ in the context comes to rescue, although not being used anywhere in the original conclusion. Thus, unlike other calculi where it is possible to drop unused variables in the context and still have a valid typing statement, in $\lambda_I^{\forall}$, it is not always possible. In the example above dropping $a : A$ from the context sabotages the subtyping relation $\forall x : A. \mathbb{N} \to \mathbb{N} \leq \mathbb{N} \to \mathbb{N}$ we rely on to conclude the typing relation. (unless we can find another inhabitant for type $A$ somehow).

However, since $\lambda_I^{\forall}$ supports fixpoints, all monotypes in our system are easily inhabited with diverging programs as inhabitants (recall that, due to the issue discussed in Section 4.1, our fixpoints only support monotypes). For instance, we do not need the variable $a$ in the context in the example above, we can just find the instantiation $\mu\, x : A.\ x$ to satisfy rule S-FORALL-L for any monotype $A$. Moreover, it is *likely* that any polytype is inhabited with the help of monotype fixpoints. For example, type $\forall A : \star. A \to B$ with $B$ being an abstract type, has an inhabitant $\Lambda A : \star. \lambda\, x : A.\ \mu\, y : B.\ y$.

Nevertheless, since polymorphic types in our system also serve as "implicit function types", it would be weird to infer diverging programs as their implicit arguments. Inferring infinite loops is not a big deal for the time being, because we guarantee the runtime irrelevance of all the implicit arguments. But this design will not be reasonable if we are going to relax the restriction of runtime irrelevance of implicit parameters in the future. Furthermore, it is also reasonable to consider similar language designs without fixpoints (for instance, if our goals are to develop theorem provers or strongly normalizing languages). In this case, the approach of finding inhabitants by creating non-terminating terms would not be possible. Thus such an approach would not be very generalizable.

*Occurrences of variables that matter.* Another issue for the strengthening is that even if we have the issue of inhabitability covered, it is tricky to guarantee that a variable unused in the final subtyping conclusion, does not matter for the whole derivation. For example, in rule S-APP, where type $A$ in the premises does not occur at all in the conclusion, we cannot guarantee that a variable not occurring in the conclusion, does not occur in type $A$. We can construct the following example:

$$\frac{T : \star \vdash 42 : \forall y : T \to T. \mathbb{N} \qquad T : \star \vdash \lambda\, x : \mathbb{N}.\ x : (\forall y : T \to T. \mathbb{N}) \to \mathbb{N}}{T : \star \vdash (\lambda\, x : \mathbb{N}.\ x)\ 42 : \mathbb{N}}\ \text{S-APP}$$

Here the variable $T$ does not occur in the conclusion (other than in the context, of course), but occurs in the premises. The interesting part is that although it occurs in the premises, without it, the conclusion holds with a different derivation tree (by directly deriving $\vdash 42 : \mathbb{N}$). For this example, although the strengthening property holds, we do not currently know how to determine whether a variable really *matters* for a derivation in general, so that we can safely remove

it from the context.

So for the two issues we discussed above, we decide to live with the lack of strengthening property in our system for now, at the cost of a slightly more complicated system with additional kinding premises in rule S-FORALL-R and the addition of rule S-FORALL.

## 6. Related Work

*Implicit Dependent Type Calculus.* Implicit polymorphism in a dependently typed setting is discussed in the implicit calculus of constructions (ICC) (Miquel, 2001) and ICC*(Barras and Bernardo, 2008). The ICC features generalized polymorphic types and typing rules to express the idea of implicit instantiation. The ICC does not explicitly have a subtyping relation between polymorphic types. Therefore the expressiveness of reasoning between polymorphic types is limited to top-level polymorphic types. Like in $\lambda_I^{\forall}$, implicit parameters does not impact the runtime semantics of the ICC.

Implicit function types in ICC* are not interpreted as polymorphic function types. The main focus is on the distinction between implicit functions (universal types and implicit abstraction) and explicit functions (Π-types and lambda abstraction). The typing rules about the implicit part and explicit part of the language mirror each other. The generalization and instantiation aspect of the implicit function types are not featured. ICC* depends on its transformation to ICC to obtain type safety of the language, therefore the parameters of implicit functions have no impact on runtime behaviour as well.

*Type-inference and unification with dependent types.* There has been little work on formalizing type inference for calculi with dependent types, although essentially all implementations of theorem provers or dependently typed languages perform some form of type-inference. One important challenge for type inference in systems with dependent types and a conversion rule is that they require *higher-order unification*, which is known to be undecidable (Goldfarb, 1981). The *pattern* fragment (Miller, 1991b) is a well-known decidable fragment. Much literature on unification for dependent types (Reed, 2009; Abel and Pientka, 2011; Gundry and McBride, 2013; Cockx et al., 2016; Ziliani and Sozeau, 2015; Coen, 2004) is built upon the pattern fragment. Algorithms for type-inference used in Agda and (Dependent) Haskell have been described and formalized to some degree in various theses (Norell, 2017; Gundry, 2013; Eisenberg, 2016). However, as far as we know there is not a clear specification and complete metatheory (let alone mechanized) for such algorithms.

The current GHC Haskell's language of types and kinds is already dependently typed, but has no type conversion. Thus it is able to avoid higher-order unification. Recent work by Xie et al. (2020) describes algorithms and specifications for the form of (dependently typed) kind-inference currently present in GHC Haskell. The dependently typed language of types and kinds is closely related to $\lambda_I^{\forall}$. In particular in both calculi type equality is based only on $\alpha$-equivalence. One difference is that in GHC Haskell and, more precisely, in the

core language employed by GHC, there are no type-level lambdas. The GHC Haskell source language does allow type families (Chakravarty et al., 2005), which mimic type-level functions. However, type families, unlike lambda functions, are not first class, and do not support partial application. They are encoded in terms of equality constraints, casts and mechanisms similar to those employed by type classes. There is some work to make type-level functions provided type families first-class (Kiss et al., 2019) and also partially applied, but this still does not enable full type-level lambdas (see also the discussion in Section 8.1 of Kiss et al. (2019) for more details). In our work we do allow type-level lambdas but lambdas can only be equal up to $\alpha$-equivalence. Another difference is that the kind-inference system formalized by Xie et al. is not higher-ranked like ours. In this way Xie et al. manage to avoid the mutual dependency issue that we have in our polymorphic subtyping relation.

*Type-inference for higher-ranked polymorphism.* Type-inference for *higher-ranked polymorphism* (HRP) (Dunfield and Krishnaswami, 2013; Le Botlan and Rémy, 2003; Leijen, 2008; Vytiniotis et al., 2008; Peyton Jones et al., 2007; Serrano et al., 2018; Odersky and Läufer, 1996; Zhao et al., 2019) extends the classic Hindley-Milner algorithm (Hindley, 1969; Milner, 1978; Damas and Milner, 1982), removing the restriction of top-level (let) polymorphism only. Type inference for HRP aims at providing inference for System F-like languages. In particular existing HRP approaches allow *synthesis of type arguments* and use type annotations to aid inference, since type-inference for full System F is well-known to be undecidable (Wells, 1999).

The work on HRP is divided into two strands: *predicative* HRP (Dunfield and Krishnaswami, 2013; Peyton Jones et al., 2007; Odersky and Läufer, 1996; Dunfield and Krishnaswami, 2019; Zhao et al., 2019) and *impredicative* HRP (Le Botlan and Rémy, 2003; Leijen, 2008; Vytiniotis et al., 2008; Serrano et al., 2018). In predicative HRP instantiations can only synthesize monotypes, whereas in impredicative HRP there's no such restriction. However, impredicative HRP is quite complex because the polymorphic subtyping relation for impredicative HRP is undecidable (Tiuryn and Urzyczyn, 1996). Thus reasonable restrictions that work well in practice are still a focus of active research. The monotype restriction on predicative instantiation is considered reasonable and practical for most programs. It is currently in use by languages such as (GHC) Haskell, Unison (Chiusano and Bjarnason, 2015) and PureScript (Freeman, 2017). The original work on polymorphic subtyping by Odersky and Läufer also enforces the monotype restriction in their subtyping rules (rule $\leq \forall L$) to prevent choosing a polytype in the instantiation. Based on polymorphic subtyping as their declarative system, Dunfield and Krishnaswami (2013) (DK) develop an algorithmic system for predicative HRP type inference. DK's algorithm was manually proved to be sound, complete, and decidable. With a more complex declarative system (Dunfield and Krishnaswami, 2019), DK extended their original work with new features. Recently Zhao et al. (2019) formalized DK's type system in the Abella theorem prover.

*Dependent Types and Subtyping.* A major difficulty in languages with subtyping is that the introduction of dependent types makes typing and subtyping depend on each other. This causes several difficulties in developing the metatheory for calculi that combine dependent types and subtyping. Almost all previous work (Aspinall and Compagnoni, 1996; Zwanenburg, 1999; Castagna and Chen, 2001; Chen, 1997, 2003) attempts to address such problem by somehow *untangling* typing and subtyping, which has the benefit that the metatheory for subtyping can be developed before the metatheory of typing. Nevertheless, several results and features prove to be challenging.

Our work builds on the work done on Pure Iso-Type Systems (PITS) (Yang and Oliveira, 2019), and *unified subtyping* (Yang and Oliveira, 2017). PITS is a variant of pure type systems (PTSs), which captures a family of calculi with *iso-types*. Iso-types generalize iso-recursive types (Pierce, 2002), and provide a simple form of type casts to address the combination of recursion and dependent types. Yang and Oliveira (2017) introduce a calculus, called $\lambda_I$, supporting OOP features such as higher-order subtyping (Pierce and Steffen, 1997), bounded quantification and top types. To address the challenges posed by the combination of dependent types and subtyping, $\lambda_I$ employs unified subtyping: a novel technique that unifies typing, subtyping and well-formedness into one relation. Therefore, $\lambda_I$ takes a significantly different approach compared to previous work, which attempts to fight the entanglement between typing and subtyping. In contrast, $\lambda_I$ embraces such tangling by collapsing the typing and subtyping relations into the same relation. This approach is different from Hutchins' technique, which eliminates the typing relation and replaces it with a combination of subtyping, well-formedness and reduction relations. In contrast, unified subtyping retains the traditional concepts of typing and subtyping, which are just two particular cases of the unified subtyping relation.

Although the $\lambda_I$ calculus formalized by Yang and Oliveira shares the use of unified subtyping with $\lambda_I^\forall$, there are substantial differences between the two calculi. Most importantly, $\lambda_I$ only has explicit polymorphism via $\Pi$ types. There are no implicit functions and universal quantification ($\forall$ types) in $\lambda_I$, and also no guessing of monotypes. $\lambda_I^\forall$ supports implicit polymorphism, and guessing the monotypes used for instantiation brings significant complications, for instance for proving type safety (as discussed in Section 4.4). The subtyping rules for universal quantification (which do not exist in $\lambda_I$) also bring considerable challenges for transitivity, and the proof technique used by $\lambda_I^\forall$ differs considerably from the proof technique used in $\lambda I$. Unlike $\lambda_I$, $\lambda_I^\forall$ does not support bounded quantification, which brings some welcome simplifications to some of the unified subtyping rules. Besides these differences other differences include the use of a call-by-name semantics in $\lambda_I^\forall$ (see also the discussion in Sections 5.1 and 5.2), and the use of the $\star : \Box$ axiom in $\lambda_I^\forall$ versus the use of $\star : \star$ in $\lambda_I$.

*Dependent Types with Explicit Casts.* Another problem is the interaction between dependent types and recursion. For this problem, a general solution that has recently emerged is the use of type casts to control type-level computation. In such an approach explicit casts are used for performing type-level compu-

tations. A motivation for using type casts is to decouple strong normalization from the proofs of metatheory, which also makes it possible to allow general recursion. There have been several studies (Stump et al., 2008; Sjöberg et al., 2012; Kimmell et al., 2012; Sjöberg and Weirich, 2015; Weirich et al., 2013; van Doorn et al., 2013; Yang et al., 2016) working on using explicit casts instead of conversion rule in a dependently typed system. In $\lambda_I^{\forall}$ we adopt a simple formulation of casts based on iso-types (Yang et al., 2016), but we believe that more powerful notions of casts could work too.

*Dependent Object Types.* Dependent Object Types (DOT) (Amin et al., 2012, 2014; Rompf and Amin, 2016) is another family of systems that discusses subtyping in a dependently typed setting. Unlike the traditional dependent type systems that are based on lambda calculus, DOT embraces the idea of "everything is an object" and features the *path-dependent types*. The path-dependent type is a restricted form of dependent types. Path-dependent types support return types of functions to mention their parameters, but only member accessing operations are allowed for the "depended value", and instead of all terms, only variable names can occur in the accessing path. This restriction rules out traditional problems in dependent type systems like handling type-level computation, and allows DOT to focus more on the subtyping aspect like reasoning about type bounds. Also, thanks to this restriction, DOT can more easily separate the concept of terms and types unlike conventional dependent type systems. Therefore, the mutual dependency of typing and subtyping is also not an issue for DOT.

DOT with Implicit Functions (DIF) (Jeffery, 2019) is an interesting extension of DOT that adds implicit functions. Since path-dependent types can encode parametric polymorphism, adding implicit functions implies adding implicit polymorphism. The treatment of implicit parameters in DIF is quite similar to ICC (Miquel, 2001) in terms of the generalization (GEN) and instantiation (INST) rules shown in Section 2.2. Hence their system shares a similar constraint of being unable to handle implicits at higher-ranked positions. However in DIF, implicit arguments are runtime relevant, and can be retrieved by a special variable. This comes with a restriction that implicit arguments can only be variables in the typing context when inferred.

*Refinement Types and Manifest Systems. Manifest* systems (Greenberg et al., 2010) is one of the styles of contract-oriented programming (in contrast to the *latent* systems (Hinze et al., 2006)), where contracts (the conditions that programmers expect to satisfy) are expressed in the type system. $\lambda_H$ (Flanagan, 2006; Greenberg et al., 2010) is one calculus that includes dependent types and subtyping simultaneously. The subtyping relation expresses the implication relation between contract satisfaction conditions. Unfortunately, this brings the difficulty of potential mutual dependency between typing and subtyping. To overcome this issue, $\lambda_H$ builds another layer of denotational semantics on top of subtyping rules to avoid that subtyping depends on typing. However, this introduces other complications in their metatheory. System $\mathrm{F_H}$ (Belo et al.,

2011) and $F^{\sigma}_{H}$ (Sekiyama et al., 2017) provide another interesting idea to deal with this mutual dependency. They get rid of the subtyping aspect in their type system, but later "recover" it after the system is defined to prove the ideas expressed by subtyping hold for their systems. Sekiyama et al. (2017) called this technique the *subsumption-free formulation*. However, it is likely that such technique is difficult to apply for systems that reason about implicit polymorphism, since systems like ICC (Miquel, 2001) that mentions subtyping relations *post facto* often fail to reason about polymorphism at higher ranks.

*Feature Comparison.* Lastly we present a summary comparing $\lambda^{\forall}_{I}$ and some of the closest related calculi in the literature in Table 1. Table 2 contains the shorthand and references for the calculi used in the comparison in Table 1. We select the following features for the comparison:

- **Dependent types:** Whether the system supports dependent types.
- **Subtyping:** Whether there is a subtyping relation in the typing rules. The complexity of subtyping relations varies but we do not dive into details here.
- **Implicit arguments:** Whether some form of implicit polymorphism is supported. The monotype restriction on instantiation is assumed.
- **Relevant arguments:** Whether implicit instantiations can be runtime relevant.
- **Rank-n polymorphism:** Whether polymorphic subtyping supports higher-ranked polymorphism.
- **Mechanization:** Whether the correctness of metatheory (and algorithm, if available) is mechanically checked instead of manually.

We use ✓, if a feature is supported, × if it is not supported. The features of relevant (implicit) arguments and rank-n polymorphism only make sense in calculi with implicit arguments. Thus for calculi without implicit arguments we use – to mean "does not apply".

## 7. Conclusion

In this article, we presented a design of a dependently typed calculus called $\lambda^{\forall}_{I}$. $\lambda^{\forall}_{I}$ generalizes non-dependent polymorphic subtyping by Odersky and Läufer (Odersky and Läufer, 1996) and contains other features like general recursion and explicit casts for type-level computations. We adopt the techniques of the Unified Subtyping (Yang and Oliveira, 2017) to avoid the mutual dependency between typing and subtyping relation to simplify the formalization. Besides other relevant theorems about typing and subtyping, *transitivity* and *type safety* are proved mechanically with the Coq proof assistant.

| | ICC | $\lambda I_{\leq}$ | Zhao19 | DK19 | DH | $\lambda_I^{\forall}$ |
|---|---|---|---|---|---|---|
| Dependent types | ✓ | ✓ | × | Equality[a] | ✓ | ✓ |
| Subtyping | × | ✓ | ✓ | ✓ | × | ✓ |
| Implicit arguments | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| Relevant arguments | × | - | × | × | ✓ | × |
| Rank-n polymorphism | × | - | ✓ | ✓ | ✓ | ✓ |
| Mechanization | × | ✓ | ✓ | × | × | ✓ |

Table 1: Feature Comparison of calculi closely related to $\lambda_I^{\forall}$.

[a]Only type-level equality is dependent, other parts of the system are not.

| Shorthand | Reference/Description |
|---|---|
| ICC | Implicit Calculus of Constructions (Miquel, 2001). |
| $\lambda I_{\leq}$ | Unified Subtyping calculus by Yang and Oliveira (2017). |
| Zhao19 | Zhao et al. (2019) mechanization of DK's type system. |
| DK19 | Dunfield and Krishnaswami (2019) |
| DH | Dependent Haskell (Eisenberg, 2016). |
| $\lambda_I^{\forall}$ | Our calculus. |

Table 2: Description of the calculi in Table 1.

In the future, we will attempt to lift various restrictions that originally simplify the metatheory, such as the kind restriction on polymorphic types and the runtime irrelevance of implicit arguments. We would also like to study the impact to the metatheory of adding $\top$ types to our language, which is a common feature of a subtyping relation. Most importantly, we consider the development of a well-specified algorithmic system a major challenge in our future work. The current formulation of $\lambda_I^{\forall}$ is declarative due to the mono-expression guesses.

While $\lambda_I^{\forall}$ still has some limitations, we believe that it already includes many of the core features that are important for typed functional languages. Assuming that we have an implementation of a core language based on $\lambda_I^{\forall}$, we expect that interesting and expressive functional languages can be built on top of such core language. For instance, all the features of Haskell 98, including higher-kinds, algebraic datatypes and type classes (Kaes, 1988; Wadler and Blott, 1989) should be easily encodable in $\lambda_I^{\forall}$. Furthermore, some features not in Haskell 98, but available in modern versions of GHC Haskell, such as higher-ranked polymorphism or certain kinds of dependent types are also supported in $\lambda_I^{\forall}$. GADTs (Cheney and Hinze, 2003; Xi et al., 2003) and type-families (Chakravarty et al., 2005) are more challenging as they require a more powerful form of casts and additional support for equality. Previous work on PITS has shown how some forms of GADTs and equality can be modelled using a variant of cast operators that employ a more powerful parallel reduction relation. We believe that $\lambda_I^{\forall}$ can also employ such variant of casts, although this also remains future work.

**Acknowledgements**

We are grateful to anonymous reviewers that helped improving the presentation of our work. This work has been sponsored by Hong Kong Research Grant Council projects number 17209519 and 17209520.

**References**

R. Hindley, The principal type-scheme of an object in combinatory logic, Transactions of the American Mathematical Society 146 (1969) 29–60.

R. Milner, A theory of type polymorphism in programming, Journal of computer and system sciences 17 (1978) 348–375.

L. Damas, R. Milner, Principal type-schemes for functional programs, in: POPL '82, 1982.

M. Odersky, K. Läufer, Putting type annotations to work, in: POPL '96, 1996.

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

J. Dunfield, N. R. Krishnaswami, Complete and easy bidirectional typechecking for higher-rank polymorphism, in: ICFP '13, 2013.

P. Freeman, PureScript, 2017. URL: `http://www.purescript.org/`.

Disciple Development Team, The Disciplined Disciple Compiler, 2017. URL: `http://disciple.ouroborus.net/`.

J. Zhao, B. C. d. S. Oliveira, T. Schrijvers, A mechanical formalization of higher-ranked polymorphic type inference, in: ICFP '19, 2019.

T. Coquand, G. Huet, The calculus of constructions, Information and Computation 76 (1988) 95–120.

L. Augustsson, Cayenne — A language with dependent types, in: ICFP '98, ACM, 1998, pp. 239–250.

T. Altenkirch, N. A. Danielsson, A. Löh, N. Oury, ΠΣ: Dependent types without the sugar, in: Functional and Logic Programming, Springer, 2010, pp. 40–55.

V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. E. III, P. Fu, G. Kimmell, T. Sheard, A. Stump, S. Weirich, Irrelevance, heterogenous equality, and call-by-value dependent type systems, in: MSFP '12, 2012, pp. 112–162.

A. Stump, M. Deters, A. Petcher, T. Schiller, T. Simpson, Verified programming in Guru, in: PLPV '09, 2008, pp. 49–58.

S. Weirich, J. Hsu, R. A. Eisenberg, System FC with explicit kind equality, in: ICFP '13, ACM, 2013, pp. 275–286.

C. Casinghino, V. Sjöberg, S. Weirich, Combining proofs and programs in a dependently typed language, in: POPL '14, ACM, 2014, pp. 33–45.

V. Sjöberg, S. Weirich, Programming up to congruence, in: POPL '15, ACM, 2015, pp. 369–382.

U. Norell, Towards a practical programming language based on dependent type theory, Ph.D. thesis, Chalmers University of Technology, 2007.

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

S. Weirich, A. Voizard, P. H. A. de Amorim, R. A. Eisenberg, A specification for dependent types in Haskell, in: ICFP '17, Association for Computing Machinery, 2017.

H. Barendregt, Introduction to generalized type systems, Journal of Functional Programming 1 (1991) 125–154.

G. Kimmell, A. Stump, H. D. E. III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, K. Y. Ahn, Equational reasoning about programs with general recursion and call-by-value semantics, in: PLPV '12, 2012, pp. 15–26.

Y. Yang, X. Bi, B. C. d. S. Oliveira, Unified syntax with iso-types, in: APLAS '16, Springer, 2016, pp. 251–270.

D. Aspinall, A. Compagnoni, Subtyping dependent types, in: LICS '96, 1996, pp. 86–97.

Y. Yang, B. C. d. S. Oliveira, Unifying typing and subtyping, in: OOPSLA '17, 2017.

W. D. Goldfarb, The undecidability of the second-order unification problem, Theoretical Computer Science 13 (1981) 225–230.

A. Miquel, The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping, in: TlCA '01, Springer, 2001, pp. 344–359.

B. Barras, B. Bernardo, The implicit calculus of constructions as a programming language with dependent types, in: International Conference on Foundations of Software Science and Computational Structures, Springer, 2008, pp. 365–379.

Coq development team, The coq proof assistant, `http://coq.inria.fr/`, ????

D. Le Botlan, D. Rémy, MLF: Raising ML to the power of System F, in: ICFP '03, 2003.

D. Leijen, HMF: Simple type inference for first-class polymorphism, in: ICFP '08, 2008.

D. Vytiniotis, S. Weirich, S. Peyton Jones, FPH: First-class polymorphism for Haskell, in: ICFP '08, 2008.

Y. Yang, B. C. Oliveira, Pure iso-type systems, Journal of Functional Programming 29 (2019).

T. Æ. Mogensen, Efficient self-interpretation in lambda calculus, Journal of Functional Programming (1992) 345–364.

J. Cheney, R. Hinze, First-class phantom types, Technical Report, Cornell University, 2003.

H. Xi, C. Chen, G. Chen, Guarded recursive datatype constructors, in: POPL '03, 2003, pp. 224–235.

R. Eisenberg, Dependent Types in Haskell: Theory and Practice, Ph.D. thesis, University of Pennsylvania, 2016.

B. C. Pierce, Types and programming languages, MIT press, 2002.

P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad hoc, in: POPL '05, 1989, pp. 60–76.

S. Kaes, Parametric overloading in polymorphic programming languages, in: ESOP '88, 1988, pp. 131–144.

G. J. Sussman, G. L. Steele, Scheme: A interpreter for extended lambda calculus, Higher-Order and Symbolic Computation 11 (1998) 405–439.

P. Martin-Löf, An intuitionistic theory of types, in: G. Sambin, J. M. Smith (Eds.), Twenty-five years of constructive type theory (Venice, 1995), volume 36 of *Oxford Logic Guides*, Oxford University Press, 1998, pp. 127–172.

D. S. Hutchins, Pure subtype systems, in: POPL '10, ACM, 2010, pp. 287–298.

J. Dunfield, N. R. Krishnaswami, Lemmas and proofs for "complete and easy bidirectional typechecking for higher-rank polymorphism" (2013).

B. Accattoli, G. Guerrieri, Open call-by-value, in: APLAS '16, Springer, 2016, pp. 206–226.

D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, Journal of logic and computation 1 (1991a) 497–536.

D. Miller, Unification of simply typed lambda-terms as logic programming (1991b).

J. Reed, Higher-order constraint simplification in dependent type theory, in: LFMTP '09, ACM, 2009, pp. 49–56.

A. Abel, B. Pientka, Higher-order dynamic pattern unification for dependent types and records, in: TLCA '11, Springer, 2011, pp. 10–26.

A. Gundry, C. McBride, A tutorial implementation of dynamic pattern unification, Unpublished draft (2013).

J. Cockx, D. Devriese, F. Piessens, Unifiers as equivalences: Proof-relevant unification of dependently typed data, in: ICFP '16, ACM, New York, NY, USA, 2016, pp. 270–283. doi:10.1145/2951913.2951917.

B. Ziliani, M. Sozeau, A unification algorithm for Coq featuring universe polymorphism and overloading, in: ICFP '15, ACM, New York, NY, USA, 2015, pp. 179–191.

C. S. Coen, Mathematical knowledge management and interactive theorem proving, Ph.D. thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5, 2004.

U. Norell, Towards a practical programming language based on dependent type theory, Ph.D. thesis, Chalmers University of Technology and Göteborg University, 2017.

A. Gundry, Type Inference, Haskell and Dependent Types, Ph.D. thesis, University of Strathclyde, 2013.

N. Xie, R. Eisenberg, B. C. d. S. Oliveira, Kind inference for datatypes, in: POPL '20, 2020.

M. M. Chakravarty, G. Keller, S. P. Jones, S. Marlow, Associated types with class, in: POPL '05, 2005, pp. 1–13.

C. Kiss, T. Field, S. Eisenbach, S. Peyton Jones, Higher-order type-level programming in haskell, Proc. ACM Program. Lang. (2019).

A. Serrano, J. Hage, D. Vytiniotis, S. Peyton Jones, Guarded impredicative polymorphism, in: PLDI '18, 2018.

J. B. Wells, Typability and type checking in System F are equivalent and undecidable, Annals of Pure and Applied Logic 98 (1999) 111–156.

J. Dunfield, N. R. Krishnaswami, Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types, in: POPL '19, 2019.

J. Tiuryn, P. Urzyczyn, The subtyping problem for second-order types is undecidable, 1996.

P. Chiusano, R. Bjarnason, Unison, 2015. URL: http://unisonweb.org.

J. Zwanenburg, Pure type systems with subtyping, in: TLCA '99, 1999, pp. 381–396.

G. Castagna, G. Chen, Dependent types with subtyping and late-bound overloading, Information and Computation 168 (2001) 1–67.

G. Chen, Subtyping calculus of construction, Mathematical Foundations of Computer Science 1997 (1997) 189–198.

G. Chen, Coercive subtyping for the calculus of constructions, in: POPL '03, ACM, 2003, pp. 150–159.

B. C. Pierce, M. Steffen, Higher-order subtyping, Theoretical computer science 176 (1997) 235–282.

F. van Doorn, H. Geuvers, F. Wiedijk, Explicit convertibility proofs in pure type systems, in: LFMTP '13, ACM, 2013, pp. 25–36.

N. Amin, A. Moors, M. Odersky, Dependent object types, in: FOOL '12, ACM, 2012.

N. Amin, T. Rompf, M. Odersky, Foundations of path-dependent types, in: OOPSLA '14, ACM, 2014, pp. 233–249.

T. Rompf, N. Amin, Type soundness for dependent object types (dot), in: OOPSLA '16, ACM, 2016, pp. 624–641.

A. Jeffery, Dependent object types with implicit functions, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, 2019, pp. 1–11.

M. Greenberg, B. C. Pierce, S. Weirich, Contracts made manifest, in: POPL '10, 2010, pp. 353–364.

R. Hinze, J. Jeuring, A. Löh, Typed contracts for functional programming, in: International Symposium on Functional and Logic Programming, Springer, 2006, pp. 208–225.

C. Flanagan, Hybrid type checking, in: POPL '06, 2006, pp. 245–256.

J. F. Belo, M. Greenberg, A. Igarashi, B. C. Pierce, Polymorphic contracts, in: ESOP '11, Springer, 2011, pp. 18–37.

T. Sekiyama, A. Igarashi, M. Greenberg, Polymorphic manifest contracts, revised and resolved, ACM Transactions on Programming Languages and Systems (TOPLAS) 39 (2017) 1–36.