Mechanized Local Type-Inference with Subtyping and Dependent Types

July 13, 2021

1 Abstract

Statically typed programming languages such as Java, C#, C++, Swift or Scala are used to implement much of the software used by our society. Such software may range from simple mobile apps, to critical software used in medical devices, financial services, airplanes or cars. It is therefore paramount that the various algorithms used in the implementations of those programming languages are *correct*. In other words, programming language implementations are *critical software*, since they can ultimately be used to implement other critical software. A bug in a programming language implementation can have disastrous consequences. Moreover it can also defeat other verification techniques used to validate the correctness of software (which usually assume that the programming language implementations themselves are correct).

Type inference algorithms are an important aspect of programming language implementations. All the aforementioned statically typed programming languages have sophisticated type systems combining various features. These features include *subtyping* and *generics* (a form of parametric polymorphism with subtyping). Such type systems are quite expressive, but often they can require a heavy (type) annotation burden. To alleviate the problem of too many type annotations, many languages support some form of type-inference, which is used to automatically infer the types used in a program. Type inference maintains the burden of annotations low, enabling programmers to focus on the program logic and making programs shorter and more readable. There are a few desirable properties for a type-inference algorithm. First and foremost a type-inference algorithm should be *correct*. In this context correctness means that if a type is inferred for an expression or variable then such inferred type should indeed be a valid (correct) type. A second property that is desirable for a type inference algorithm is *predicability*: it should be easy for programmers to determine where annotations are needed. A final desirable property is *decidability*: a type inference algorithm should terminate and either infer a type, or reject the program with a type error.

The aim of this project is to study and develop type-inference algorithms for rich type systems, which include features such as *subtyping*, *generics* and forms of *dependent types*. We focus on *local type inference* techniques, which were studied and popularized by Pierce and Turner (2000) to provide a practical form of type-inference for simple languages with subtyping and polymorphism. Unfortunately, despite their pioneering work, local type inference techniques remain understudied and there is a big gap between the forms of local type-inference studied in the literature and the implementations of type-inference in languages such as Scala, Java or C#. While those implementations are loosely based on the ideas of Pierce and Turner they have various limitations and it is unclear whether they are correct and have other desirable properties. One issue is that the type systems of languages like Java and Scala are significantly more powerful and feature-rich than those studied by Pierce and Turner, therefore dealing with new features is done in an ad-hoc fashion. Another issue is that it is hard to extend the metatheory of local type inference to account for new features. This stems from the fact that such metatheory has been done by hand (i.e. using hand-written proofs and theorems) and it requires a lot of expert knowledge to start with. Therefore almost all implementations of type-inference in mainstream languages are not formally studied, leading to ad-hoc and unverified designs. There are several reports of bugs in existing implementations of type-inference algorithms, and for many languages it is quite unpredictable when annotations are necessary.

The starting point for this project is the PI's work at ICFP 2019 and POPL 2020. The ICFP 2019 work offers the first fully mechanized type-inference algorithm for *higher-ranked polymorphism*. This work uncovered several errors in previous manual proofs, and at the same time provided *machine checked* proofs for a new type-inference algorithm. This proposal will apply the same methodology to *local type-inference* and significantly extend existing formalizations of local type inference with various features of practical interest, including *bounded quantification*, *dependent types* and *union and intersection types*. Our POPL 2020 work, which shows novel techniques for type-inference in the presence of dependent types, will be relevant to extensions of local type-inference with dependent types. All our work will be mechanically formalized in existing theorem provers such as Coq or Abella. Therefore, the algorithms and metatheory will be checked up to some of the highest standards of correctness for *critical software*. Furthermore the machine checked formalizations will provide a *reusable* foundation for other researchers to build upon for their own work on type-inference. Ultimately we expect that our work will significantly advance the state-of-the-art for local type-inference, which is routinely employed in mainstream programming languages.

2 Impact and Objectives

Pathways to Impact Statement

The current type-inference algorithms employed in most mainstream languages (such as Java, C# or Scala) are loosely based on published variants of local type inference. However, work on local type inference has so far only studied very restricted subsets of languages with polymorphism and subtyping. Thus, many advanced features present in languages like Java or Scala are not accounted for in published work and are treated in an ad-hoc way in real-world implementations. As a consequence, languages with complex type systems (such as Scala or Java) commonly have bugs in their implementations of type-inference (which can be very dangerous!); are quite unpredictable as to where type annotations are required; and are overly conservative when doing type-inference since it is not clear to the compiler writers how to deal with features not previously studied in the literature.

There are two longer-term *potential* impacts of this project that can improve mainstream programming languages (like Java, C# or Scala) and consequently have an impact on the millions of programmers that use those languages. The first impact is to improve the correctness and usability of mainstream programming languages. Mainstream programming language designers will be able to adopt type-inference algorithms based on the outcomes of this project to make the type-inference in their languages more correct, robust and predictable. The second impact is to advance the state-of-the-art of local type-inference for modern type system features, which have not been studied in the past. Languages such as Scala are leading the way in terms of innovative type system features. The Scala type system includes, among other features, intersection types, union types (in the latest versions) and a simple form of dependent types. Even languages like Java have many features that have not been formally studied for local type-inference (for instance wildcards). Local type-inference for type systems with these features has received little attention. Our work will change this by doing a thorough study of type inference in the presence of many of those features. Ultimately this will allow languages like Java or Scala to improve their type-inference support for those advances features. More importantly, it will clearly document and study the algorithms so that other programming language implementations (which may not support those features yet) can then easily adopt similar language features.

Another impact of our project will be for the type-inference research community itself. In the end of the project we expect to have a highly reusable fully mechanized framework with the algorithms and metatheory for local type-inference available. There are two major benefits of having mechanized formalizations using theorem provers like Coq or Abella. Firstly, mechanized formalizations allow us to have full assurances of correctness, since the correctness of the proofs is automatically checked by a computer. In contrast manual proofs of properties are prone to human errors, which can invalidate the results. For instance, in our earlier ICFP 2019 paper, we have uncovered various errors in earlier (manual) proofs for type-inference algorithms (including lemmas that are actually false). Since proofs for programming languages tend to be long, tedious and error-prone, the use of theorem proofs is a great help to ensure the absence of mistakes. Secondly, a fully mechanized framework for local type inference will enable other researchers to more quickly and easily study their own extensions to type-inference by reusing and adapting our framework. Researchers will be able to download the sources for the proofs and easily interact with the proofs and modify them. For some language extensions, extending the framework may be as simple as adding a few new cases to some theorems. Other extensions may be more complex and involve new or modified theorems. In any case a theorem prover will assist and guide the researchers through those modifications.

Objectives

- **Mechanizing local type inference for Object-Oriented (OO) style subtyping.** This objective aims at providing, for the first time, a fully mechanized account of local type-inference for polymorphic languages with OO style subtyping.
- **Mechanizing local type inference for simple dependent types.** The second objective is to investigate local type inference in the presence of simple forms of dependent types. Languages like Scala already support simple forms of dependent types, and dependent types are becoming more and more

prominent in programming languages. They are therefore an important feature for programming languages in the near future. This objective will investigate the algorithms and metatheory for local type-inference with dependent types.

- **Mechanizing local type inference for type systems with intersection and union types.** The final objective is to extend the results in the first objective with two important and non-trivial features: intersection and union types. Intersection and union types are employed currently in various languages (such as Scala, RedHat Ceylon, Facebook Flow or Microsoft Typescript). This goal will investigate the algorithms and metatheory for local type-inference with those features.

3 Background of Research

This section motivates local type-inference in the presence of subtyping and dependent types, and its challenges. Relevant existing work by the PI and others in this area is also discussed.

Motivation Modern mainstream languages such as Java or Scala support a combination of *subtyping*, *parametric polymorphism* (or *generics*) and *first-class functions* (or *lambdas*). Such features enable writing code such as the following snippet of Java:

```
List<String> numbers = Arrays.asList("1", "2", "3", "4", "5", "6");
List<Integer> even = numbers.stream()
                                .map(s -> Integer.valueOf(s))
                                .filter(number -> number % 2 == 0)
                                .collect(Collectors.toList());
```

This code processes a list of strings representing numbers, converts the strings into numbers and filters the even numbers in the list. Thus the list `even` is `[2,4,6]`. This piece of elegant code is made practical by a form of local type-inference, which is helpful in two ways. Firstly, type inference enables *local synthesis of type arguments*, which means that applications of generic methods (i.e. methods parametrized by some types) will *automatically* infer the type arguments of the method. An example of local synthesis is the method call `map(s -> Integer.valueOf(s))` above. The `map` method from the `Stream<T>` class has the following type:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

The method takes a function that transforms objects of (a supertype of) type `T` (the type of elements of the input stream) into objects of (a subtype of) type `R`. `map` applies the function to all the elements in the stream, thus producing a stream of type `Stream<R>`. Note that the type `R` is *parametrized*. Therefore uses of `map` must instantiate `R` with a concrete class. In `map(s -> Integer.valueOf(s))` we have that `R = Integer`, which is determined automatically by local synthesis. Secondly, local type inference employs *bidirectional type-checking techniques* to propagate known type-information. For example, in the snippet above the type of `numbers` is `List<String>` and we know the type of the `map` method. Therefore in `map(s -> Integer.valueOf(s))` we can deduce that `s` is of type `String`. Without such form of local type-inference the code above would need to explicitly provide the types for instantiation and the types of arguments for the lambda functions, making the code cluttered with type annotations.

Unfortunately, as noted by Smith and Cartwright [1], the type-inference algorithms in languages like Java are *broken*: i.e. there are cases where the algorithms incorrectly synthesize instantiations, and also cases where the algorithms cannot find an instantiation. In other words the algorithms are neither *sound* nor *complete*. As they point out the new features in Java 5 (and later versions of Java) were "*so novel that its technical foundations, particularly with regard to type inference, had not yet been thoroughly investigated in the research literature*". Our proposal aims at studying local type-inference at a foundational level. We have two main aims: 1) to revisit classic local type-inference type algorithms and mechanize them in a theorem prover; 2) to study many language features that were not previously studied in the research literature of local type inference (including *bounded quantification* or simple forms of *dependent types*).

3.1 Work Done by Others

Local type-inference Local type inference was originally proposed by Pierce and Turner [2]. It consists of the combination of two techniques: *bi-directional type-checking* and *local synthesis of type arguments*. In their pioneering work Pierce and Turner considered a language based on System F [3, 4], but extended with a top and bottom types and a corresponding subtyping relation. This language does not account for bounded quantification. Thus it is weaker than System $F_{<:}$ [5], which is a minimal calculus that deals with *bounded quantification*. Various works extended local type-inference over the years to make it more

powerful (i.e. to require less type annotations), but often the method has been applied to small variants of the language originally considered by Pierce and Turner. For instance, Odersky et al. proposed *colored local type inference* [6], which is the inspiration for Scala's type inference algorithm [7], and refined local type inference for *explicit* polymorphism by propagating partial type information. Their work still considers the same language as Pierce and Turner, but extended with records. Pierce and Turner [2] briefly consider an extension with bounded quantification, sketching an algorithm and some proofs, but a more thorough study of type inference with bounded quantification is desirable. Castagna et al. [8], studies one of the most advanced forms of local type inference that accounts for union and intersection types, but their algorithm is restricted to first-order polymorphism. As far as we know, there are no attempts in the literature at mechanically formalizing the algorithms and meta-theory of local type inference in theorem provers.

Global type-inference for higher-ranked polymorphism Type-inference for *higher-ranked polymorphism* (HRP) [9, 10, 11, 12, 13, 14, 15] extends the classic Hindley-Milner algorithm [16, 17, 18], removing the restriction of top-level (let) polymorphism only. Type inference for HRP aims at providing inference for System F-like languages, and is closely related to Pierce and Turner's local type-inference. Both type inference techniques 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 [19]. The main difference is that HRP type inference targets a System F-like language without subtyping, whereas the local type inference targets a System F-like language with subtyping. As Pierce and Turner argue, global type type inference techniques (which employ long-distance constraints such as unification variables) interact poorly with subtyping, which motivates their (local) method of using only information from adjacent nodes in the syntax tree.

The work on HRP is divided into two strands: *predicative* HRP [9, 13, 15, 20] and *impredicative* HRP [10, 11, 12, 14]. 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 [21]. Thus reasonable restrictions that work well in practice are still a hot topic in 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 [22] and PureScript [23]. Most relevant for this proposal is Dunfield and Krishnaswami's (DK) [9] algorithm for predicative HRP type inference. DK's algorithm was manually proved to be sound, complete and decidable in 70 pages of manual proofs. With a more complex declarative system [20], DK extended their original work with new features and developed a manual proof of over 150 pages. Though carefully written, some of DK's proofs are incorrect as illustrated by our ICFP 2019 work [24].

MLSub A recent breakthrough in the area of (global) type-inference for type systems with subtyping is MLSub [25]. MLSub extends the Hindley-Milner type system with support for subtyping. A key innovation of MLSub is that it has compact principal types, which had been a challenge in previous research on type-inference in the presence of subtyping [26, 27, 28]. MLSub is significantly more ambitious than local type-inference, and requires no annotations (in the tradition of Hindley-Milner). However MLSub does not account for HRP and its algorithms and metatheory have not been mechanically formalized.

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 is that type inference for many systems with dependent types requires *higher-order unification*, which is known to be undecidable [29]. The *pattern* fragment [30] is a well-known decidable fragment. Much literature on unification for dependent types [31, 32, 33, 34, 35, 36] 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 [37, 38, 39]. However as far as we know there is not a clear specification and complete metatheory (let alone mechanized) for such algorithms. In the *implicit calculus of constructions* [40] an elegant (but purely declarative) curry-style version of the calculus of constructions is presented. As the author suggests type-inference for this calculus without annotations is undecidable.

Mechanical Formalizations of Type-Inference Algorithms Mechanizing certain algorithmic aspects of type inference, such as unification and constraint solving, has received very little attention and is still challenging. Part of the difficulty is that many practical type-inference algorithms require more complex binding structures with output contexts or various forms of constraint solving procedures. These are hard to mechanize in a theorem prover. Algorithm $\mathcal{W}$, was formally verified [41] in Isabelle/HOL [42]. The treatment of new variables was tricky at that time, while the overall structure follows the structure of Damas's manual proof closely. Later on, other researchers [43, 44] formalized algorithm $\mathcal{W}$ in Coq [45]. Nominal techniques [46] in Isabelle/HOL have been developed to help programming language formalizations, and are used for a similar verification [47]. Moreover, Garrigue [48] mechanized a type inference algorithm, with the help of locally nameless [49], for Core ML extended with structural polymorphism and recursion.

3.2 Work Done by the PI

Dr. Bruno C. d. S. Oliveira will be the PI in this project. His research interests are type systems, object-oriented programming and functional programming. He has 6 paper awards in the area of programming languages, including 2 awards at top conferences: the *best paper award* for ECOOP 2012 [50]; and a *distinguished paper award* for ICFP 2019 [24]. His work had a direct impact on existing widely used compilers. His OOPSLA 2010 [51] work on Scala implicits helped popularizing the feature, and his later work on formal models of implicits [52] influenced the Scala designers to adopt first-class implicits. The implementation of the GHC Haskell feature of *quantified class constraints* is based on the PI's 2017 Haskell Symposium paper [53]. Finally, his work on *kind inference for datatypes* [54] provides a formalization of kind inference for a sophisticated language of datatypes similar to that used by modern GHC Haskell, and will likely influence future improvements of GHC kind inference.

Relevant Related Work Of particular relevance for this project is the PI's recent work at ICFP 2019 [24] and POPL 2020 [54]. The ICFP 2019 work presents a fully mechanized formalization of predicative HRP type inference in the Abella theorem prover. The work on this proposal will adapt various of the techniques developed in the ICFP 2019 work, to deal with various new type system features, and corresponding mechanized algorithms for type inference. The POPL 2020 paper presents a kind inference algorithm for a very rich dependently typed language of datatypes that represents a subset of the features present in the GHC Haskell compiler. This work will be very relevant for Objective 2, since it provides a number of techniques that can address important problems in type inference for dependent types. The PI's OOPSLA 2017 work on unifying typing and subtyping [55] will also be relevant for Objective 2. This work shows a novel technique that simplifies the design of languages combining subtyping and dependent types.

4 Research Plan and Methodology

Our proposal consists of three objectives. The first objective aims at mechanizing local type inference for polymorphic languages with OO style subtyping. The second objective aims at mechanizing local type inference for simple dependent types. The final objective is to mechanize local type inference for type systems with intersection and union types. To accomplish the 3 objectives, the project will require two graduate students (RA1 and RA2). Section 3a) of the proposal application provides a detailed description of the student tasks. Section 2b)(ii) provides a timeline for the research activities.

4.1 Objective 1: Mechanizing local type inference for OO subtyping

Our first objective is to revisit Pierce and Turner's local type-inference, mechanize the new algorithms and metatheory in a theorem prover and extend the results with bounded quantification. Since Pierce and Turner first proposed their technique for local type-inference in the year 2000, much work has been done in type

inference for HRP [9, 13, 24]. The new developments HRP algorithms provide many useful techniques that can improve the local type inference state-of-the-art. We will exploit such developments in this objective.

Preliminary Results: Declarative and algorithmic subtyping with $\top$ and $\bot$ Our ICFP 2019 work [24] presents declarative and algorithmic systems for HRP type-inference, but does not deal with common OO subtyping features such as top and bottom types. Based on the ICFP work, we have mechanized the declarative and algorithmic systems for a System F-like calculus with subtyping (including $\top$ and $\bot$ types) in the Abella theorem prover. We proved some basic results about well-formedness and subtyping already, but the main theorems (soundness, completeness and decidability) are still being proved. Compared to the ICFP 2019 work, there are a few notable differences. Firstly, in the same spirit as local type inference, we make constraint solving localized to subtyping/instantiation, so that the type system itself does only simple bi-directional type-checking. In practice this means that local type-inference is much more modest when doing inference of functions without explicit types for the arguments when compared to global type inference approaches used in existing HRP algorithms. Secondly, we need to deal with subtyping for features such as a top and a bottom type appropriately, which is where the main technical challenges lie.

Figure 1 shows the *syntax*, *declarative subtyping* and *algorithmic subtyping* used in our Abella formalization. The syntax of (declarative) types includes the unit type, universal quantification, function types, type variables and top and bottom types. Additionally algorithmic types can also have existential variables $\widehat{\alpha}$, which play a similar role to unification variables. The declarative subtyping relation extends Odersky and Läufer's subtyping relation for polymorphic types [15]. The novel rules are the standard rules for subtyping of top and bottom types, namely $\leq$`Top` and $\leq$`Bot`. Following the ICFP 2019 work, the algorithmic system is based on judgment worklists (Γ), which unify traditional (ordered) typing contexts and judgements. However, there is a new kind of entry ($\Gamma, L \leq \widehat{\alpha} \leq U$) in the worklists. This new entry serves the purpose of tracking subtyping constraints on type variables, keeping a list of types L, which are lower bounds, and another list U, which are upper bounds.

An algorithm with late instantiation checks While adding top and bottom types to the declarative system is straightforward, the algorithmic system becomes quite different in nature compared to the algorithm used in the ICFP 2019 work. Most importantly, the introduction of $\top$ and $\bot$ types means that the algorithm *cannot perform eager instantiation of existential type variables*. Consider a function `f` with the type:

```
f<a> : a -> a -> Bool
```

Note that in `f` the syntax `<a>` denotes a generic type parameter, which can be implicitly instantiated. In existing systems with HRP type inference (or even Hindley-Milner), it is safe to conclude that the partial application `f 3` has type `Int -> Bool` and thus the instantiation used in this application is `a = Int`. Unfortunately, in the presence of a $\top$ type, such conclusion is not valid in general. For instance, the full application `f 3 x` where `x :` $\top$ should be valid. But this can only be true if `a` is instantiated with type $\top$. In the specification of subtyping $\top \to \top \to \mathit{Bool} <: \mathit{Int} \to \top \to \mathit{Bool}$ has a valid derivation. Consequently $\forall a.a \to a \to \mathit{Bool} <: \mathit{Int} \to \top \to \mathit{Bool}$ is also valid (with the choice $a = \top$). Doing early instantiation ($a = \mathit{Int}$ as in previous HRP algorithms) would prevent the instantiation $a = \top$ and would not lead to any valid subtyping derivation for this example. In other words, it is not enough to rely only on partial information regarding instantiation to conclude what the correct type for instantiation is. Instead all the information is needed to pick a safe instantiation type in the presence of features such as $\top$ and $\bot$ types. Early instantiation would lead to an *incomplete algorithm*.

Therefore, the algorithm presented in Figure 1 does not do eager substitutions once an existential variable is matched up with some concrete monotype. Instead, the monotypes are added to the set of subtyping constraints for the existential variable (i.e. updating the L and U bounds associated with an existential variable $\widehat{\alpha}$ in the environment). Only after all the subtyping constraints are collected, a check is done (in the second rule of the algorithm) that validates whether all the constraints are valid. Essentially this rule compares all the collected lower bounds l_n with all the collected upper bounds u_m for subtyping. For space reasons it is not possible to discuss in detail the other aspects of the algorithm, but the main change compared to the ICFP work is the (necessary) late instantiation check for existential variables.

Future work: Soundness, completeness and decidability While we have proved basic results about the subtyping relation, we have not yet proved key results about the algorithmic subtyping relation. In particular the most important results are *soundness*, *completeness* and *decidability*. Soundness will show that any valid derivations in the algorithmic system are also valid in the declarative type system. We expect soundness to be the easiest of the results to show. The main challenge is how to relate the algorithmic derivations with the declarative derivations. In order to do so we will employ an auxiliary transfer relation inspired by the ICFP 2019 metatheory. The main complication is that we now have to deal with the subtyping constraint entries ($\Gamma, L \leq \widehat{\alpha} \leq U$) when doing the transfer. *Completeness* will also rely on the auxiliary transfer relation. However we expect that this result is going to be harder to prove due to the significant changes in the algorithmic relation and the late instantiation checks. This will require the development of some auxiliary metatheory that enables proving completeness. Finally, *decidability* will also require different meta-theory, and in particular, different size measures. The second rule of the algorithmic system is the main challenge. This rule introduces many new judgements on the right side to be processed. For decidability we must show however, that the introduction of these judgements decreases some size measure. Indeed the key observation here is that the number of existential variables decreases (despite the number of judgements increasing). We expect that translating this idea into a formal statement will be key to proving decidability.

Future work: Local and global type systems In order to be useful for type-inference, the subtyping relation needs to be complemented with a type system. For local type-inference the type system itself should be relatively straightforward as there is no unification process going on: only simple bi-directional type-checking propagation of type information is needed. One small challenge is that on applications the type system still needs to guess the type instantiations. To deal with this challenge we expect to use techniques from Xie and Oliveira's *application mode* [56], which provides an alternative mechanism for bi-directional type-checking. Using the application mode we can reuse the subtyping relation as a mechanism for instantiation. We also want to explore the possibilities for *global* type-inference algorithms. This will likely borrow some ideas from MLSub [25]. The key difference and challenge is that MLSub only allows for *let polymorphism* whereas we allow HRP. *Let polymorphism* allows simplifying the scope of unification variables (which can be global), but with HRP the scope of variables needs to be carefully controlled.

Future work: Bounded quantification One extension that we want to investigate is *bounded quantification* [5], which is a feature that exists in essentially every mainstream OO language with generics. Pierce and Turner [57] sketch some algorithms and proofs for local type inference in the presence of bounded quantification, but there are several limitations. One limitation of their system is that inter-dependent bounds, such as $\forall(X <: \top, Y <: X).\ S \to T$ (for some S and some T), are not supported. Furthermore, sometimes it is not possible to find a best type for instantiation. We will revisit the problem of local type inference with bounded quantification, find simple declarative and algorithmic formulations, attempt to address the limitations of Pierce and Turner's algorithms and mechanically formalize the metatheory in a theorem prover.

4.2 Objective 2: Local type inference for dependent types

Modern OO languages like Scala include restricted forms of dependent types (more precisely *dependent object types*) [58]. Dependent types are also a widely popular feature in modern functional languages, such as Idris [59], Agda [60] or modern GHC Haskell [54]. Some form of type-inference is also important on those languages to make their use practical. However type-inference in the presence of dependent types has proved extremely challenging, as there are numerous technical difficulties. In this objective we will study local type-inference for various (restricted) forms of dependent types with practical applications.

Preliminary work: A specification of local synthesis for simple dependent types As a starting point we started investigating the problem of local type-inference for a language similar to the *calculus of constructions* [61] (but with the $\star : \star$ axiom and some restrictions). As part of our preliminary work we designed a specification for a calculus that supports local synthesis of arguments. There two main challenges that we faced in the design of this calculus, which lead to important design decisions.

The first challenge is due to the conversion rule typically present in dependently typed languages. In dependently typed languages the conversion rule enables type-level computation and it is widely used in languages like Agda or Idris. Unfortunately it also introduces major complications for type-inference. A well-known result is that type-inference for systems with a conversion rule requires *higher-order unification*, which is known to be *undecidable* [29]. Therefore to avoid higher-order unification we simply drop the conversion rule in our initial calculus. Nevertheless there are still practical applications for languages with such a restriction. For instance, GHC Haskell language of types and kinds (which is dependently typed) has no type conversion and it is indeed quite closely related to the language that we plan to study first. Indeed our recent POPL 2020 paper [54] describes algorithms and specifications for (the very ambitious form) of type-inference currently present in GHC Haskell. In GHC Haskell there are no type-level lambdas and thus no type conversion (although type-level computation can be achieved by other means [62]).

The second challenge is that formulating subtyping for dependently typed languages is tricky due to the mutual dependency between various relations (typing, subtyping and well-formedness). Subtyping is commonly used to specify a *most general than* relation between types. It is for instance used by DK [9] and our own ICFP 2019 work. However the mutual dependencies that arise in a dependently typed setting create various technical problems for the metatheory. For instance, unlike non-dependently typed languages, it is not possible to develop the metatheory of subtyping separately from typing. This problem has been observed by multiple researchers in previous work [63, 64, 65, 66], with multiple solutions making different compromises. Our OOPSLA 2017 work [55] is relevant here as it proposes a simple solution for this problem. Following the same spirit as *Pure Type Systems* [67], which attempt to unify syntax and the typing and well-formedness relations, our OOPSLA 2017 work proposes to go one step further: it shows how to unify typing and subtyping into a single relation. This solves the problem of dependencies in that now there is only a single relation that depends on itself. Furthermore it results on a compact specification compared to a variant with multiple independent relations.

Figure 2 shows the syntax and the unified subtyping relation for the calculus. Notably we distinguish between two kinds of universal quantification. Π-types are the usual dependent function spaces, which must be given explicit arguments when applied. In contrast $\forall$-types allow implicit instantiation are akin to universal quantifiers with implicit parameters in languages like Coq and Agda. Monotypes are just a subset of types without $\forall$-types. The form of the unified subtyping relation is $\Gamma \vdash e_1 \leq e_2 : A$. This relation means that e_1 is a subtype of e_2 and both expressions have type A. The form of the relation follows the unified subtyping idea in the OOPSLA 2017 paper. Like in that work, typing is just a special case when the two expressions are syntactically the same. The relation itself is inspired by the (declarative) polymorphic subtyping relation for System F by Odersky and Läufer [15]. An important difference is that, due to the more general dependently typed setting, we must track the types during the subtyping relation. Tracking types is important here because we must guess monotypes of the correct type/kind. This is materialized in the rule PS-FORALLL, which requires a monotype t of type A. This rule also shows the entanglement between typing and subtyping: the subtyping relation requires typing in order to validate whether the monotype is of the right type. In contrast, for System F, the subtyping relation only operates on types, and therefore does not need to track types in order to guess monotypes. Thus subtyping is completely independent of typing in that setting. The rule PS-FORALLR generalizes another of the rules for polymorphic subtyping by Odersky's and Läufer. The rule PS-FORALL allows the introduction of implicit arguments. The remaining rules are essentially simplified versions of the rules in the OOPSLA 2017 paper. The main source of simplification is that in that work considers bounded quantification [5], which is not accounted for here. Finally we note that in our POPL 2020 work avoids the entanglement between typing and subtyping by employing a very limited form of subtyping, which is less expressive than the form proposed here.

Future work: Metatheory and algorithmic version As initial future work we plan to study the metatheory for this calculus and formalize it in a theorem prover. At the bottom of Figure 2 there are some properties that we expect to hold for this calculus. Similar properties were proved in the OOPSLA 2017 work. More importantly, we also plan to study sound and complete algorithmic versions of the relation. Currently the relation is declarative for two reasons: 1) it guesses monotypes, rather than providing a method to

compute the monotypes; 2) it is not syntax directed due to the subsumption rule PS-SUB. We expect that the algorithmic version will be challenging due to the more complex set of rules (compared to Odersky and Laufer's system) and the fact that types need to be tracked in the relation. To make the system syntax directed we will adapt bi-directional typechecking techniques to the unified subtyping setting.

Future work: Further extensions We will consider two extensions. The first extension is the addition of $\top$ and $\bot$ types. This will connect with Objective 1 and we will try to adapt the results of Objective 1 into the more complex dependently typed setting here. The second extension is how to add type-level computation. As explained before type-level computation can be quite challenging to add due to potential issues with higher-order unification. Therefore we first plan to introduce explicit casts for type-level computation [68, 69], which provides an alternative to the conversion rule. With casts equality is still based on α-equivalence (rather than β-equivalence in systems with the conversion rule). Because of that we expect that first-order unification is enough. Some of the techniques that we have explored in our POPL 2020 work will be relevant here, since that system is also based on α-equivalence. Finally, we are also interested in studying systems with the conversion rule (and based on β-equivalence). Because of the undecidability that arizes from higher-order unification in such a setting we will look at *sound*, but *not complete* algorithms. We expect existing work on *pattern unification* [70] to be relevant for this endeavour.

4.3 Objective 3: Local type inference with union and intersection types

Union and intersection types are a common feature in many modern OOP languages, including Scala, RedHat Ceylon, Facebook Flow and Microsoft TypeScript. Unfortunately, despite their prominence in widely used languages, there is little work on type-inference in the presence of these features. In this objective we will extend the results in the Objective 1 with intersection and union types, and investigate the algorithms and metatheory for local type-inference with those features.

Preliminary work: A declarative subtyping relation with union and intersection types It is not hard to define a *declarative* version of a subtyping relation for implicit polymorphism in the presence of intersection and union types. Figure 3 shows such a relation. In essence we can extend Odersky and Läufer's subtyping relation with standard subtyping rules for intersection and union types. For instance, rule $\leq And$ states that a type A is a subtype of an intersection $B\&C$ if A is both a subtype of B and C. This relation can be shown to be reflexive and transitive. However, the harder challenges (as often is the case with type inference) are not in the declarative version, but in the algorithmic version. We discuss future work next.

Future work: Metatheory and algorithmic version The big challenge is to define the algorithmic system and the corresponding metatheory (soundness, completeness and decidability). One interesting aspect of the algorithmic system with union and intersection types is that, while the system in Objective 1 has to use lists of types L and U to track lower and upper bounds, we expect to be able to use intersections to model multiple lower bounds and unions to model upper bounds. One source of complexity in adding intersection and union types is that many rules are overlapping, which will add complexity to the already complex metatheory of the algorithmic formulation. Perhaps more problematic is the interaction between instantiation and intersection and union types, which has not received much attention before.

Future work: Distributivity Some programming languages, including Ceylon, Julia and the research language SEDEL [71], also feature distributivity rules that allow subtyping derivations such as $(Int \to Char)\&(Bool \to String) <: Int\&Bool \to Char\&String$. This derivation is allowed because intersections distribute over function types. While distributivity is a useful feature, it brings major complications. In particular the algorithmic versions of subtyping become highly non-trivial, and have been the object of study of several recent research work [72, 73, 74]. In previous work we have managed to find simple subtyping algorithms with distributivity of intersections [72]. However none of the previous works has investigated the interaction between distributivity and instantiation, which is likely to be likely non-trivial and will add yet more complexity to the algorithmic version of subtyping. We will extend our previous algorithm with distributivity of intersections with union types and instantiation and study the corresponding metatheory.

References

[1] Smith, D., Cartwright, R.: Java type inference is broken: Can we fix it? In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. OOPSLA '08, New York, NY, USA, ACM (2008) 505–524

[2] Pierce, B.C., Turner, D.N.: Local type inference. ACM Trans. Program. Lang. Syst. **22**(1) (January 2000)

[3] Reynolds, J.C.: Towards a theory of type structure. In: Proceedings of the 'Colloque sur la Programmation'. Number 19, Paris, France (1974) 408–425

[4] Girard, J.Y.: Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. PhD thesis, Université Paris VII (1972)

[5] Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system f with subtyping. Information and Computation **109**(1-2) (1994) 4–56

[6] Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '01 (2001)

[7] Plociniczak, H.: Decrypting Local Type Inference. PhD thesis, École Polytechnique Fédérale de Lausanne (2016)

[8] Castagna, G., Nguyen, K., Xu, Z., Abate, P.: Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15 (2015)

[9] Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP '13 (2013)

[10] Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of system F. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. ICFP '03 (2003)

[11] Leijen, D.: HMF: Simple type inference for first-class polymorphism. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08 (2008)

[12] Vytiniotis, D., Weirich, S., Peyton Jones, S.: FPH: First-class polymorphism for Haskell. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08 (2008)

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

[14] Serrano, A., Hage, J., Vytiniotis, D., Peyton Jones, S.: Guarded impredicative polymorphism. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2018 (2018)

[15] Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '96 (1996)

[16] Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society **146** (1969) 29–60

[17] Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3) (1978) 348–375

[18] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '82 (1982)

[19] Wells, J.B.: Typability and type checking in system F are equivalent and undecidable. Annals of Pure and Applied Logic **98**(1-3) (1999) 111–156

[20] Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. Proc. ACM Program. Lang. **3**(POPL) (January 2019)

[21] Tiuryn, J., Urzyczyn, P.: The subtyping problem for second-order types is undecidable. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. (1996)

[22] Chiusano, P., Bjarnason, R.: Unison (2015)

[23] Freeman, P.: PureScript (2017)

[24] Zhao, J., Oliveira, B.C.d.S., Schrijvers, T.: A mechanical formalization of higher-ranked polymorphic type inference. Proc. ACM Program. Lang. **3**(ICFP) (July 2019)

[25] Dolan, S., Mycroft, A.: Polymorphism, subtyping, and type inference in mlsub. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017 (2017)

[26] : Type inference for recursively constrained types and its application to oop. Electronic Notes in Theoretical Computer Science **1** (1995) 132 – 153 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[27] Trifonov, V., Smith, S.F.: Subtyping constrained types. In: Proceedings of the Third International Symposium on Static Analysis. SAS '96 (1996)

[28] Fran cois, P.: Type inference in the presence of subtyping: from theory to practice. PhD thesis, Universite Paris 7 (1998)

[29] Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science **13**(2) (1981) 225–230

[30] Miller, D.: Unification of simply typed lambda-terms as logic programming. (1991)

[31] Reed, J.: Higher-order constraint simplification in dependent type theory. In: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ACM (2009) 49–56

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

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

[34] Cockx, J., Devriese, D., Piessens, F.: Unifiers as equivalences: Proof-relevant unification of dependently typed data. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. ICFP 2016, New York, NY, USA, ACM (2016) 270–283

[35] Ziliani, B., Sozeau, M.: A unification algorithm for coq featuring universe polymorphism and overloading. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015, New York, NY, USA, ACM (2015) 179–191

[36] Coen, C.S.: Mathematical knowledge management and interactive theorem proving. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5 (2004)

[37] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Göteborg University (2017)

[38] Gundry, A.: Type Inference, Haskell and Dependent Types. PhD thesis, University of Strathclyde (2013)

[39] Eisenberg, R.: Dependent Types in Haskell: Theory and Practice. PhD thesis, University of Pennsylvania (2016)

[40] Miquel, A.: The implicit calculus of constructions. In: TLCA. LNCS, Springer (2001) 344–359

[41] Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm W in Isabelle/HOL. Journal of Automated Reasoning **23**(3) (1999) 299–318

[42] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283. Springer Science & Business Media (2002)

[43] Dubois, C.: Proving ML type soundness within Coq. Theorem Proving in Higher Order Logics (2000) 126–144

[44] Dubois, C., Menissier-Morain, V.: Certification of a type inference tool for ML: Damas–Milner within Coq. Journal of Automated Reasoning **23**(3) (1999) 319–346

[45] Coq development team: The coq proof assistant. `http://coq.inria.fr/`

[46] Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4) (2008) 327–356

[47] Urban, C., Nipkow, T.: Nominal verification of algorithm W. From Semantics to Computer Science. Essays in Honour of Gilles Kahn (2008) 363–382

[48] Garrigue, J.: A certified implementation of ML with structural polymorphism and recursive types. Mathematical Structures in Computer Science **25**(4) (2015) 867–891

[49] Charguéraud, A.: The locally nameless representation. Journal of Automated Reasoning **49**(3) (Oct 2012) 363–408

[50] Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: ECOOP'12. (2012)

[51] Oliveira, B.C., Moors, A., Odersky, M.: Type classes as objects and implicits. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '10

[52] d. S. Oliveira, B.C., Schrijvers, T., Choi, W., Lee, W., Yi, K.: The implicit calculus: A new foundation for generic programming. In: PLDI 2012: 33rd ACM Conference on Programming Language Design and Implementation. (2012)

[53] Bottu, G.J., Karachalias, G., Schrijvers, T., Oliveira, B.C.d.S., Wadler, P.: Quantified class constraints. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. Haskell 2017 (2017)

[54] Ningning Xie, R.E., d. S. Oliveira, B.C.: Kind inference for datatypes. In: POPL 2020: Symposium on Principles of Programming Languages. (2020)

[55] Yang, Y., Oliveira, B.C.d.S.: Unifying typing and subtyping. In: OOPSLA '17. (2017)

[56] Xie, N., Oliveira, B.C.d.S.: Let arguments go first. In Ahmed, A., ed.: Programming Languages and Systems, Cham, Springer International Publishing (2018) 272–299

[57] Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1) (2000) 1–44

[58] Rompf, T., Amin, N.: Type soundness for dependent object types. In: OOPSLA'16. (2016)

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

[60] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)

[61] Coquand, T., Huet, G.: The calculus of constructions. Information and Computation **76** (1988)

[62] Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming. ICFP '05 (2005)

[63] Aspinall, D., Compagnoni, A.: Subtyping dependent types. In: LICS '96. (1996) 86–97

[64] Chen, G.: Coercive subtyping for the calculus of constructions. In: POPL '03, ACM (2003) 150–159

[65] Hutchins, D.S.: Pure subtype systems. In: POPL '10, ACM (2010) 287–298

[66] Zwanenburg, J.: Pure type systems with subtyping. In: TLCA '99. (1999) 381–396

[67] Barendregt, H.: Introduction to generalized type systems. Journal of Functional Programming **1**(2) (1991) 125–154

[68] Sjöberg, V., Weirich, S.: Programming up to congruence. In: POPL '15, ACM (2015) 369–382

[69] Yang, Y., Bi, X., Oliveira, B.C.d.S.: Unified syntax with iso-types. In: APLAS '16, Springer (2016) 251–270

[70] Miller, D.: Unification of simply typed lambda-terms as logic programming. In Furukawa, K., ed.: Eighth International Logic Programming Conference, Paris, France, MIT Press (June 1991)

[71] Bi, X., d. S. Oliveira, B.C.: Typed first-class traits. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018. (2018)

[72] Bi, X., d. S. Oliveira, B.C., Schrijvers, T.: The essence of nested composition. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands. (2018) 22:1–22:33

[73] Muehlboeck, F., Tate, R.: Empowering union and intersection types with integrated subtyping. Proc. ACM Program. Lang. **2**(OOPSLA) (October 2018)

[74] Chung, B., Nardelli, F.Z., Vitek, J.: Julia's efficient algorithm for subtyping unions and covariant tuples (pearl). In: 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. (2019) 24:1–24:15

5 Figures

(Syntax)

Types	A,B,C	$::=$	$1 \mid \cancel{\top \mid \bot} \mid a \mid \forall x.A \mid A \to B \mid \widehat{\alpha}$
Mono-types	τ,σ,l,u	$::=$	$1 \mid \cancel{\top \mid \bot} \mid a \mid \tau_1 \to \tau_2 \mid \widehat{\alpha}$
Declarative Context	Ψ	$::=$	$\cdot \mid \Psi, a$
Bound Collection	L,U	$::=$	$\{\overline{\tau}\}$
Worklist	Γ	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, L \le \widehat{\alpha} \le U \mid \Gamma \Vdash A \le B$

$\boxed{\Psi \vdash A \le B}$ *(Declarative Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash a \le a}\ \le\texttt{Var} \qquad \frac{}{\Psi \vdash 1 \le 1}\ \le\texttt{Unit} \qquad \frac{}{A \le \top}\ \le\textbf{Top} \qquad \frac{}{\bot \le A}\ \le\textbf{Bot}$$

$$\frac{\Psi \vdash B_1 \le A_1 \quad \Psi \vdash A_2 \le B_2}{\Psi \vdash A_1 \to A_2 \le B_1 \to B_2}\ \le\to \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \le B}{\Psi \vdash \forall a.\, A \le B}\ \le\forall\texttt{L} \qquad \frac{\Psi, b \vdash A \le B}{\Psi \vdash A \le \forall b.\, B}\ \le\forall\texttt{R}$$

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ'. *(Algorithmic Subtyping)*

$$\Gamma, a \longrightarrow \Gamma$$

$$\Gamma, L \le \widehat{\alpha} \le U \longrightarrow \Gamma \Vdash l_1 \le u_1 \Vdash l_1 \le u_2 \Vdash \ldots \Vdash l_n \le u_m \qquad L = \{l\}_n, U = \{u\}_m$$

$$\cancel{\Gamma \Vdash A \le \top \longrightarrow \Gamma}$$

$$\cancel{\Gamma \Vdash \bot \le B \longrightarrow \Gamma}$$

$$\Gamma \Vdash 1 \le 1 \longrightarrow \Gamma$$

$$\Gamma \Vdash a \le a \longrightarrow \Gamma$$

$$\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \le \widehat{\alpha} \longrightarrow \Gamma$$

$$\Gamma \Vdash A_1 \to A_2 \le B_1 \to B_2 \longrightarrow \Gamma \Vdash A_2 \le B_2 \Vdash B_1 \le A_1$$

$$\Gamma \Vdash \forall a.A \le B \longrightarrow \Gamma, \bot \le \widehat{\alpha} \le \top \Vdash [\widehat{\alpha}/a]A \le B$$

$$\Gamma \Vdash A \le \forall b.B \longrightarrow \Gamma, b \Vdash A \le B$$

$$\Gamma[L \le \widehat{\alpha} \le U] \Vdash \widehat{\alpha} \le A \to B \longrightarrow \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, L \le \widehat{\alpha} \le U \cup \{\widehat{\alpha}_1 \to \widehat{\alpha}_2\}] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \le A \to B$$
$$\text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)$$

$$\Gamma[L \le \widehat{\alpha} \le U] \Vdash A \to B \le \widehat{\alpha} \longrightarrow \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, L \cup \{\widehat{\alpha}_1 \to \widehat{\alpha}_2\} \le \widehat{\alpha} \le U] \Vdash A \to B \le \widehat{\alpha}_1 \to \widehat{\alpha}_2$$
$$\text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)$$

$$\Gamma[a][L \le \widehat{\beta} \le U] \Vdash a \le \widehat{\beta} \longrightarrow \Gamma[a][L \cup \{a\} \le \widehat{\beta} \le U]$$

$$\Gamma[a][L \le \widehat{\beta} \le U] \Vdash \widehat{\beta} \le a \longrightarrow \Gamma[a][L \le \widehat{\beta} \le U \cup \{a\}]$$

$$\Gamma[L \le \widehat{\beta} \le U] \Vdash 1 \le \widehat{\beta} \longrightarrow \Gamma[L \cup \{1\} \le \widehat{\beta} \le U]$$

$$\Gamma[L \le \widehat{\beta} \le U] \Vdash \widehat{\beta} \le 1 \longrightarrow \Gamma[L \le \widehat{\beta} \le U \cup \{1\}]$$

$$\Gamma[\widehat{\alpha}][L \le \widehat{\beta} \le U] \Vdash \widehat{\alpha} \le \widehat{\beta} \longrightarrow \Gamma[\widehat{\alpha}][L \cup \{\widehat{\alpha}\} \le \widehat{\beta} \le U]$$

$$\Gamma[\widehat{\alpha}][L \le \widehat{\beta} \le U] \Vdash \widehat{\beta} \le \widehat{\alpha} \longrightarrow \Gamma[\widehat{\alpha}][L \le \widehat{\beta} \le U \cup \{\widehat{\alpha}\}]$$

Soundness: If $\Gamma \Vdash A \le B \longrightarrow^* \cdot$, then $\Gamma \vdash A \le B$.

Completeness: If $\Gamma \vdash A \le B$, then $\Gamma \Vdash A \le B \longrightarrow^* \cdot$.

Decidability: For all well-formed Γ, it is decidable whether $\Gamma \longrightarrow^* \cdot$ or not.

Figure 1: Worklist Subtyping Algorithm.

(Syntax)

Expressions	e, A, B, C	$::=$	$x \mid \star \mid e_1\, e_2 \mid \lambda x : A.\ e \mid \Pi x : A.\ B \mid \forall x : A.\ B \mid n \mid e_1 + e_2 \mid \mathit{Int}$
Contexts	Γ	$::=$	$\varnothing \mid \Gamma, x : A$
Mono-types	t	$::=$	$\star \mid x \mid \mathit{Int} \mid \Pi x : t_1.\ t_2$

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \leq x : A}\ \text{PS-VAR} \qquad \frac{}{\Gamma \vdash n \leq n : \mathit{Int}}\ \text{PS-LIT} \qquad \frac{}{\Gamma \vdash \star \leq \star : \star}\ \text{PS-STAR} \qquad \frac{}{\Gamma \vdash \mathit{Int} \leq \mathit{Int} : \star}\ \text{PS-INT}$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash e_1 \leq e_2 : B}{\Gamma \vdash \lambda x.\ e_1 \leq \lambda x.\ e_2 : \Pi x : A.\ B}\ \text{PS-ABS} \qquad \frac{\Gamma \vdash A_2 \leq A_1 : \star \qquad \Gamma, x : A_1 \vdash B_1 : \star \qquad \Gamma, x : A_2 \vdash B_1 \leq B_2 : \star}{\Gamma \vdash \Pi x : A_1.\ B_1 \leq \Pi x : A_2.\ B_2 : \star}\ \text{PS-PI}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e_1 \leq e_2 : \Pi x : A.\ B}{\Gamma \vdash e_1\, e \leq e_2\, e : B[x \mapsto e]}\ \text{PS-APP} \qquad \frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash e_1 \leq e_2 : B}{\Gamma \vdash e_1 \leq e_2 : \forall x : A.\ B}\ \text{PS-FORALL}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B[x \mapsto t] \ \leq C : \star}{\Gamma \vdash \forall x : A.\ B \leq C : \star}\ \text{PS-FORALLL} \qquad \frac{\Gamma \vdash B : \star \qquad \Gamma, x : B \vdash A \leq C : \star}{\Gamma \vdash A \leq \forall x : B.\ C : \star}\ \text{PS-FORALLR}$$

$$\frac{\Gamma \vdash e_1 \leq e_2 : A \qquad \Gamma \vdash A \leq B : \star}{\Gamma \vdash e_1 \leq e_2 : B}\ \text{PS-SUB}$$

Syntactic Sugar $\qquad \Gamma \vdash e : A \triangleq \Gamma \vdash e \leq e : A$

Reflexivity: If $\Gamma \vdash e_1 \leq e_2 : A$, then $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$.
Weakening: If $\Gamma_1, \Gamma_3 \vdash e_1 \leq e_2 : A$ and $\vdash \Gamma_1, \Gamma_2, \Gamma_3$, then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash e_1 \leq e_2 : A$.
Narrowing: If $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leq e_2 : C$, $\Gamma_1 \vdash A \leq B : \star$ and $\vdash \Gamma_1, x : A, \Gamma_2$, then $\Gamma_1, x : A, \Gamma_2 \vdash e_1 \leq e_2 : C$.
Consistency of Typing: If $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_1 \leq e_2 : B$, then $\Gamma \vdash e_1 \leq e_2 : A$.
Transitivity: If $\Gamma \vdash e_1 \leq e_2 : A$ and $\Gamma \vdash e_2 \leq e_3 : A$, then $\Gamma \vdash e_1 \leq e_3 : A$.
Substitution: If $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \leq e_2 : A$ and $\Gamma_1 \vdash e_3 : B$, then $\Gamma_1, \Gamma_2[x \mapsto e_3] \ \vdash e_1[x \mapsto e_3] \ \leq e_2[x \mapsto e_3] \ : A[x \mapsto e_3]$.
Correctness of Types: If $\Gamma \vdash A \leq B : C$, then $\Gamma \vdash C : \star$.

Figure 2: Local Type Inference for Dependent Type System

$\boxed{\Psi \vdash A \leq B}$ *(Declarative Subtyping with Intersection and Union Types)*

$$\frac{a \in \Psi}{\Psi \vdash a \leq a} \leq \texttt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1} \leq \texttt{Unit} \qquad \frac{}{A \leq \top} \leq \texttt{Top} \qquad \frac{}{\bot \leq A} \leq \texttt{Bot}$$

$$\frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2} \leq\!\to \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\ A \leq B} \leq\!\forall\texttt{L} \qquad \frac{\Psi, b \vdash A \leq B}{\Psi \vdash A \leq \forall b.\ B} \leq\!\forall\texttt{R}$$

$$\frac{\Psi \vdash A \leq B \quad \Psi \vdash A \leq C}{\Psi \vdash A \leq B \,\&\, C} \leq \texttt{And} \qquad \frac{\Psi \vdash A \leq C}{\Psi \vdash A \,\&\, B \leq C} \leq \texttt{AndL} \qquad \frac{\Psi \vdash B \leq C}{\Psi \vdash A \,\&\, B \leq C} \leq \texttt{AndR}$$

$$\frac{\Psi \vdash A \leq C \quad \Psi \vdash B \leq C}{\Psi \vdash A \mid B \leq C} \leq \texttt{Or} \qquad \frac{\Psi \vdash A \leq B}{\Psi \vdash A \leq B \mid C} \leq \texttt{OrL} \qquad \frac{\Psi \vdash A \leq C}{\Psi \vdash A \leq B \mid C} \leq \texttt{OrR}$$

Figure 3: Declarative Subtyping with Intersection and Union Types.