Local Contextual Type Inference

XU XUE, University of Hong Kong, China
CHEN CUI, University of Hong Kong, China
SHENGYI JIANG, University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, University of Hong Kong, China

Type inference is essential for programming languages, yet complete and global inference quickly becomes undecidable in the presence of rich type systems like System F. Pierce and Turner proposed local type inference (LTI) as a scalable, partially annotated alternative by relying on information local to applications. While LTI has been widely adopted in practice, there are significant gaps between theory and practice, with its theory being underdeveloped and specifications for LTI being complex and restrictive.

We propose *Local Contextual Type Inference*, a principled redesign of LTI grounded in contextual typing—a recent formalism which captures type information flow. We present *Contextual System F* (F_c), a variant of System F with implicit and first-class polymorphism. We formalize F_c using a declarative type system, prove soundness, completeness, and decidability, and introduce matching subtyping as a bridge between declarative and algorithmic inference. This work offers the first mechanized treatment of LTI, while at the same time removing important practical restrictions and also demonstrating the power of contextual typing in designing robust, extensible and simple to implement type inference algorithms.

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

Additional Key Words and Phrases: Type Inference, Contextual Typing, Local Type Inference

ACM Reference Format:
Xu Xue, Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2026. Local Contextual Type Inference. *Proc. ACM Program. Lang.* 10, POPL, Article 11 (January 2026), 30 pages. https://doi.org/10.1145/3776653

1 Introduction

Type inference plays a crucial role in programming languages, offering mechanisms that range from completing partial type information to inferring all types within a program. Type inference can be either global or local. Complete type inference, where all types can be inferred without explicit annotations, employs global algorithms with long-distance constraints, typically based on unification [Herbrand 1930; Robinson 1965]. The **Hindley-Milner (HM) type system** [Milner 1978] provides a canonical example of complete type inference. HM algorithms have a certain degree of complexity due to these advanced constraint solving techniques. Nevertheless, complete type inference leads to a simple *specification* that shields us from those algorithmic details and clearly determines which programs are accepted by the type system. For instance, the core inference rule for function application in the declarative specification of HM is notably concise and standard:

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

Authors' Contact Information: Xu Xue, University of Hong Kong, Hong Kong, China, xxue@cs.hku.hk; Chen Cui, University of Hong Kong, Hong Kong, China, ccui@cs.hku.hk; Shengyi Jiang, University of Hong Kong, Hong Kong, China, syjiang@cs.hku.hk; Bruno C. d. S. Oliveira, University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

ACM 2475-1421/2026/1-ART11
https://doi.org/10.1145/3776653

Combined with other rules, this rule enables polymorphic functions to be applied with *implicit* instantiation of type arguments. For example, given an identity function id of type $\forall\alpha.\ \alpha \to \alpha$, the application id 1 is easily inferred to have type Int under HM rules. In an *explicitly* polymorphic calculus, such as System F [Girard 1972; Reynolds 1974], the same application would require programmers to explicitly pick the instantiation and write id @Int 1.

Nevertheless the expressive power of HM is limited, as it supports only rank-1 polymorphism where all universal quantifiers appear at the top level. A consequence of this restriction is that the instantiated types cannot themselves be polymorphic. Therefore, a natural extension of the HM type system is a variant of *implicit* System F [Mitchell 1988], which supports first-class polymorphism. A well-known and standard way to formalize instantiation in first-class polymorphism is with subtyping using the following instantiation rule [Mitchell 1988; Odersky and Läufer 1996]:

$$\frac{\Gamma \vdash C \quad \Gamma \vdash [C/\alpha]A <: B}{\Gamma \vdash \forall\alpha.\ A <: B}$$

This rule relates polymorphic types with their instantiations. In the id 1 example, the polymorphic type $\forall\alpha.\ \alpha \to \alpha$ of id is a subtype of $\mathsf{Int} \to \mathsf{Int}$ by instantiating α with Int. Therefore, id can be used as a function of type $\mathsf{Int} \to \mathsf{Int}$. In contrast to HM, this rule even allows *impredicative* instantiation, where C itself is a polymorphic type. For instance, the explicit type application id @$(\forall\beta.\ \beta \to \beta)$ id is encodable in implicit System F as id id by picking C to be $\forall\beta.\ \beta \to \beta$ in the rule above.

Unfortunately, complete type inference can easily become undecidable when extended to richer type systems, including implicit System F. In fact, even for the subtyping relation itself, the instantiation rule without restrictions results in an undecidable problem [Chrząszcz 1998; Tiuryn and Urzyczyn 1996]. Therefore, in order to preserve decidability, we must impose some restrictions and turn to partial type inference schemes where some degree of user annotations is required.

Local type inference (LTI) [Pierce and Turner 2000] is a partial type inference approach, which disallows long-distance constraints and relies only on information local to adjacent nodes in the abstract syntax tree. LTI trades some of the expressive power afforded by techniques such as unification for scalability to advanced type system features. Pierce and Turner showed that advanced features such as subtyping and impredicativity are both supported with LTI. The key algorithmic idea in LTI is the use of *matching* [Huet 1976], instead of unification. Unlike unification, which poses fundamental problems related to decidability in the presence of advanced type system features, matching remains decidable even in the presence of advanced type features such as forms of subtyping, intersection types and first-class polymorphism [Düdder et al. 2013; Stirling 2009].

While programs using LTI often require more annotations compared to programs that adopt global type inference techniques, in practice the burden of annotations is still relatively low. Therefore, due to its moderate annotation burden and good scalability, LTI techniques are widely adopted in practice and are the main type inference technique adopted by many mainstream programming languages, including Java, C#, TypeScript, Flow and Scala.

Despite the pervasive use of LTI in modern programming languages, its theoretical foundations remain surprisingly underdeveloped, creating a significant disparity between theory and practice. While there have been some follow-up works [Jenkins and Stump 2018; Odersky et al. 2001] to Pierce and Turner's original work, these pale in comparison with research on global type inference, which has seen many developments over the years [Botlan and Rémy 2003; Dolan and Mycroft 2017; Dunfield and Krishnaswami 2013; Jones et al. 2007; Parreaux et al. 2024; Serrano et al. 2020]. The lack of development of LTI is partly due to the underlying theory of LTI being complex and perhaps even the perception that the approach is somewhat ad-hoc. Existing specifications for LTI require significant complexity compared to specifications for HM and implicit System F, which nicely capture the key aspects of implicit instantiation via the application rule and the subtyping

instantiation rule. Specifications for LTI often rely on monolithic application rules with complex side-conditions to specify instantiation, and they often have important restrictions compared to practical implementations. Such complexity transposes into the algorithms as well and interferes with the addition of new features. Many new features require non-trivial changes, which may explain the lack of progress in the theoretical development of LTI. Existing specifications of LTI also make it hard to compare and make a precise connection to implicit System F and other type inference algorithms for System F [Botlan and Rémy 2003; Jones et al. 2007; Leijen 2008].

This work employs *contextual typing* [Xue and Oliveira 2024], a recent generalization of bidirectional typing [Dunfield and Krishnaswami 2022; Pierce and Turner 2000], for naturally specifying and designing local type inference and their algorithms. Contextual typing extends bidirectional typing by propagating not only known type information, but also other contextual details about the surrounding context of terms [Xue and Oliveira 2024]. Contextual typing is especially suitable to specify partial type inference algorithms, since it employs *contextual type assignment systems* (CTASs) to model the contextual type information available for a term to type-check. A CTAS provides a type system specification that precisely indicates where annotations are required. Contextual typing also supports a powerful form of **contextual subsumption**, that exploits partially known contextual information [Xue and Oliveira 2024]. This enhanced expressiveness allows many other type rules within the system to remain modular, simple and largely untouched, as the complexity is managed more effectively by the contextual subsumption mechanism itself.

By leveraging contextual typing, we aim to provide a foundational approach that simplifies the specification of LTI, removes some important restrictions and facilitates the development of robust and intuitive type-inference algorithms. We concretely illustrate our approach with a variant of implicit System F, called Contextual System F (F_c). Furthermore, we believe that our work provides a solid foundation for future research addressing the gaps in terms of type system features between LTI theory and practice. In summary our contributions are:

- **Local Contextual Type Inference**, offering a simple and modular specification of LTI (Sec. 3), aligned with design choices found in mainstream programming language implementations.
- We define **matching subtyping** to specify the use of matching for constraint solving and aid in establishing the equivalence between the CTAS specification and the algorithm (Sec. 4).
- We present **Contextual System F ($\mathbf{F_c}$)**, a variant of implicit System F, demonstrating the soundness, completeness, and the decidability between its CTAS and the algorithmic rules. The algorithmic system and metatheory are presented in Sec. 5.
- All results, including the properties of Contextual System F and matching subtyping, have been **formally proven and mechanized in theorem provers**[1], ensuring their correctness and reliability. As far as we know we are the first to mechanize local type inference algorithms.
- We have a **prototype implementation** available at https://github.com/juniorxxue/LCTI and an accompanying artifact [Xue et al. 2025], which can run all the examples presented in this paper.

2 Local Type Inference and Contextual Typing

This section introduces the key ideas in local type inference and characterizes programs accepted by local (contextual) type inference, informally. Some of the examples that we use are adapted from Botlan and Rémy [2003]. We also introduce relevant background on contextual typing.

2.1 Characterizing Local Type Inference

We start by informally characterizing programs accepted/rejected by LTI by example. All the examples shown in this subsection are typeable/untypeable both in F_c and Pierce and Turner's

[1]Most of the formalization is done in Agda, but decidability is done in Rocq due to better support for arithmetic reasoning.

original formulation of LTI with one caveat: some of the programs would need to be written in an uncurried style in Pierce and Turner's approach. We will ignore this caveat here and present all programs in a curried style (directly accepted by F_c). Sec. 2.2 discusses this caveat in detail.

Local type inference. Local type inference [Pierce and Turner 2000] is a *partial* type inference technique consisting of two ideas: *local synthesis of type arguments* and *bidirectional type-checking*. Synthesis of type arguments allows programmers to write polymorphic function applications without explicitly providing type arguments. For example LTI can type-check id 1, just like the example we have seen in Sec. 1 for HM and implicit System F. Bidirectional type-checking is a technique that allows the type information to be propagated, and permits the programmer to omit certain annotations. For example, if f has the type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$, then f $(\lambda \mathsf{x}.\ \mathsf{x})$ is a well-typed term and $(\lambda \mathsf{x}.\ \mathsf{x})$ is checked against the type $\mathsf{Int} \to \mathsf{Int}$, propagated from the input type of f. This idea of type information flowing from functions to arguments has evolved into the default application rule of bidirectional typing, which has been adopted in many type systems:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \mathsf{App}$$

In bidirectional type systems there are two typing modes: $\Gamma \vdash e \Rightarrow A$ is the inference mode, and it means that the type A is inferred from analyzing e; $\Gamma \vdash e \Leftarrow A$ is the checking mode, and it means that given type A we can check whether e has type A. The flow of type information from functions to arguments is clear in the bidirectional rule above.

Top-level annotations are usually required. Unlike HM and other global type inference techniques, in LTI top-level annotations are usually required. For instance, the HM term let succ = $(\lambda \mathsf{x}.\ \mathsf{x} + 1)$ in succ 1 is not directly typeable in LTI, since there is no inference rule for lambda terms. LTI only supports checking a lambda term against a type. Therefore, either annotations are required to type-check a lambda, or the type information must come from the surrounding context (as in the f $(\lambda \mathsf{x}.\ \mathsf{x})$ example above). With LTI we can write instead:

$$\mathsf{let\ succ} : \mathsf{Int} \to \mathsf{Int} = (\lambda \mathsf{x}.\ \mathsf{x} + 1)\ \mathsf{in\ succ\ 1}$$

Alternatively, we could annotate the variable in the lambda $(\lambda \mathsf{x} : \mathsf{Int}.\ \mathsf{x} + 1)$. Moreover, unlike HM, there is no let generalization. Thus, top-level polymorphic functions must be annotated.

Implicit impredicative instantiations. One of the motivations for LTI is to support implicit impredicative polymorphism, which allows instantiations with polymorphic types without explicit type arguments. For example, LTI will instantiate the type variable α of choose to be $\forall \alpha.\ \alpha \to \alpha$:

$$\mathsf{choose} : \forall \alpha.\ \alpha \to \alpha \to \alpha, \mathsf{id} : \forall \alpha.\ \alpha \to \alpha \vdash \mathsf{choose\ id} \Rightarrow (\forall \alpha.\ \alpha \to \alpha) \to (\forall \alpha.\ \alpha \to \alpha)$$

Explicit impredicative instantiations. Like System F, LTI also supports explicit impredicative instantiation by providing an explicit type argument. For example, consider the application:

$$\mathsf{auto} : (\forall \alpha.\ \alpha \to \alpha) \to (\forall \alpha.\ \alpha \to \alpha) \vdash \mathsf{choose\ id\ auto}$$

LTI only supports a restricted form of subtyping and rejects this example as it considers the types of the two arguments $(\forall \alpha.\ \alpha \to a$ and $(\forall \alpha.\ \alpha \to \alpha) \to \forall \alpha.\ \alpha \to \alpha)$ to be unrelated. In such cases, we can explicitly instantiate id with a type application using id $@(\forall \alpha.\ \alpha \to \alpha)$ to adjust the type of id, thereby allowing the annotated expression to be accepted:

$$\vdash \mathsf{choose\ (id}\ @(\forall \alpha.\ \alpha \to \alpha))\ \mathsf{auto} \Rightarrow (\forall \alpha.\ \alpha \to \alpha) \to (\forall \alpha.\ \alpha \to \alpha)$$

Application-triggered implicit instantiation. In LTI, implicit instantiation is triggered by applications to arguments *only*. This contrasts with global type inference algorithms, where instantiation can be triggered without explicit applications. For example, $\mathsf{f} : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \vdash \mathsf{f\ id}$ is not directly accepted with LTI, since id is not applied to any argument. This design has important consequences. In some cases, more annotations or η-expansions are needed to type programs with

LTI. For instance, with f id we can η-expand id, and LTI will accept the application f (λx. id x). In other cases, however, it is possible to avoid ambiguities that other approaches must address. For example, single id does not have a most general type if both single and id can be instantiated. This expression can be typed as either $[\forall\alpha.\ \alpha \to \alpha]$ or $\forall\alpha.\ [\alpha \to \alpha]$. Due to this ambiguity, some type inference algorithms [Serrano et al. 2020] will reject this program. On the other hand, LTI will unambiguously infer the former result as implicit instantiation occurs only in applied terms:

$$\mathsf{single} : \forall\alpha.\ (\alpha \to [\alpha]) \vdash \mathsf{single\ id} \Rightarrow [\forall\alpha.\ \alpha \to \alpha]$$

Shallow instantiations. In LTI, all instantiations are *shallow*. This means that the instantiation happens only at the quantifiers in rank-1 positions in types. Note that nested rank-1 positions are also included, so the following is allowed:

$$\mathsf{h} : \mathsf{Bool} \to \forall\alpha.\ \alpha \to \alpha \vdash \mathsf{h\ true\ 1} \Rightarrow \mathsf{Int}$$

with α being implicitly instantiated to Int. Quantifiers appearing in higher-rank positions are not instantiated. For example, annotating f (of type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$) with the type $(\forall\alpha.\ \alpha \to \alpha) \to \mathsf{Int}$ will not type-check in LTI and F_c, since the subtyping statement $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \leq (\forall\alpha.\ \alpha \to \alpha) \to \mathsf{Int}$ will not trigger instantiation of the rank-2 quantifier, and subtyping will consider these two types unrelated and will fail. In several other type inference algorithms, which adopt *deep* instantiation [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996], the type $(\forall\alpha.\ \alpha \to \alpha) \to \mathsf{Int}$ is accepted and α in the input position is instantiated to Int.

2.2 Local Type Inference Specification and its Limitations

Local type inference has an interesting approach to instantiation. However, the specification proposed by Pierce and Turner is complex, and has important restrictions compared to practical implementations. A first complication is the departure from standard System F syntax regarding terms and types. Functions are, by default, *uncurried*: type arguments (if present) and function arguments are all provided at once, and *polymorphic types* are merged with function types having the syntactic form $\forall\overline{\alpha}.\ \overline{A} \to B$. We list other complications and practical limitations below.

Specifying applications and instantiation. Instantiation is monolithically modeled as part of the application rules. There are *four* application rules in total. Two of them are checking rules, and two others are inference rules. We show the inference rules for explicit and implicit instantiation next:

$$\frac{\Gamma \vdash f \Rightarrow \forall\overline{\alpha}.\ \overline{B} \to C \quad \Gamma \vdash \overline{e} \Leftarrow [\overline{A}/\overline{\alpha}]\overline{B}}{\Gamma \vdash f[\overline{A}](\overline{e}) \Rightarrow [\overline{A}/\overline{\alpha}]C}\ \text{S-App} \qquad \frac{\Gamma \vdash f \Rightarrow \forall\overline{\alpha}.\ \overline{B} \to C \quad \Gamma \vdash \overline{e} \Rightarrow \overline{D} \quad |\overline{\alpha}| > 0 \quad \Gamma \vdash \overline{D} <: [\overline{A}/\overline{\alpha}]\overline{B} \quad \forall\overline{F}.(\Gamma \vdash \overline{D} <: [\overline{F}/\overline{\alpha}]\overline{B}\ \text{implies}\ \Gamma \vdash [\overline{A}/\overline{\alpha}]C <: [\overline{F}/\overline{\alpha}]C)}{\Gamma \vdash f(\overline{e}) \Rightarrow [\overline{A}/\overline{\alpha}]C}\ \text{S-App-InfSpec}$$

The checking rules are similar and largely duplicate the logic of the inference rules. The rule for explicit instantiation (rule S-App) can be viewed as a generalization of the standard bidirectional typing rule presented earlier, except that it deals with uncurried applications and their explicit instantiations. The number of type variables $\overline{\alpha}$ can be zero, and in that case it falls back to the non-polymorphic bidirectional function application. Rule S-App-InfSpec is used for function applications with implicit instantiation, where type arguments are not required. Unlike the first rule, it requires that arguments are always *inferable* ($\Gamma \vdash \overline{e} \Rightarrow \overline{D}$). This rule first infers the types for *both* the function and its arguments, and then uses subtyping to compare the inferred type of the arguments with the input type of the function, but substituted with the *guessed* type arguments. There are a few important conditions in this rule. Firstly, the rule ensures that it is used in a polymorphic context ($|\overline{\alpha}| > 0$). The last condition enforces that the guessed type is the most precise type. Finally the condition $\Gamma \vdash \overline{D} <: [\overline{A}/\overline{\alpha}]\overline{B}$ specifies how to find solutions for instantiation using subtyping.

Notably, the type $\overline{B}$ has free type variables that must be replaced with guessed instantiations ($\overline{A}$), but no instantiations are needed for the subtype $\overline{D}$. This suggests that matching is sufficient to implement the solving process. We will come back to the topic of matching later in Sec. 4.

Uncurried applications and locality. In Pierce and Turner's formulation of LTI uncurried applications play an important role, since they define the notion of *locality* in the approach. The distinction between local and global type inference is that in local type inference only information from adjacent nodes in the abstract syntax is used to solve instantiations. In Pierce and Turner's LTI this means that all instantiations must be found from looking only at the arguments of an uncurried function. Consider a constant function applied to two arguments:

$$\mathsf{const} : \forall \alpha\ \beta.\ \alpha \to \beta \to \alpha \vdash \mathsf{const\ true\ 1} \Rightarrow \mathsf{Bool}$$

Here we have two type arguments instantiated with Bool and Int for the application const true 1. In an uncurried formulation of this application, which fits with Pierce and Turner's approach, rule S-App-InfSpec can be applied (const(true, 1)). This rule would guess the instantiations and infer the type Bool. Their rule ensures that, after applying the arguments, *all* instantiations for type variables (α and β in this case) in the uncurried application are guessed.

Second-class treatment for curried functions. Curried functions can still be encoded, since the syntax does allow for nested uncurried abstractions. Here we use the syntax $\lambda\, \overline{\alpha}\, \overline{x : A}.\ e$ to denote an uncurried function with type arguments $\overline{\alpha}$ and arguments $\overline{x : A}$. In Pierce and Turner's LTI we can write a curried variant of the const function using, for example: $\mathsf{const}_2 = \lambda\alpha\ (x : \alpha).\ \lambda\beta\ (y : \beta).\ x$. Now, we can partially apply const_2 with for example $\mathsf{const}_2(1)$ to obtain the type $\forall\beta.\ \beta \to \mathsf{Int}$. This is possible because we only need to solve α (which is determined by the first argument) for the first uncurried function. However, as Pierce and Turner acknowledge, the treatment for curried functions is second class since, for a third variant of const: $\mathsf{const}_3 = \lambda\alpha\ \beta\ (x : \alpha).\ \lambda(y : \beta).\ x$, the applications $\mathsf{const}_3(1)$ and $\mathsf{const}_3(1)(\mathsf{true})$ would both fail. The difference to const_2 is that now β is part of the first uncurried function, but it can only be determined by the second argument (y). It is not surprising that the partial application $\mathsf{const}_3(1)$ fails. However, more surprisingly, the application of the two arguments ($\mathsf{const}_3(1)(\mathsf{true})$) fails as well. The reason for this is that rule S-App-InfSpec requires that all type arguments are solved by looking *only* at the arguments of the same uncurried function, but true is an argument of another uncurried function.

Hard-to-synthesize arguments. A key limitation in Pierce and Turner's LTI approach is the so-called hard-to-synthesize arguments problem [Hosoya and Pierce 1999]. In rule S-App-InfSpec *all* arguments need to be inferable. However, this is problematic for applications of higher-order functions where the arguments are lambdas. For example, we would like to have:

$$\mathsf{twice} : \forall\alpha.\ \alpha \to (\alpha \to \alpha) \to \alpha \vdash \mathsf{twice\ 1\ (}\lambda\mathsf{x.\ x)} \Rightarrow \mathsf{Int}$$

But, if this application is uncurried (twice $(1, \lambda$x. x)) and uses rule S-App-InfSpec then we would try to infer the type of λx. x. Unfortunately, inference for lambdas is not supported in LTI and consequently this expression fails to type-check. Interestingly enough, if twice is curried (with the lambda argument being part of a different uncurried function as described in the previous paragraph), then we could type-check twice since the argument 1 would be sufficient to instantiate α and then rule S-App could be used to check the lambda. This example actually illustrates that writing uncurried functions does not always maximize the chances of inference: sometimes a curried function type-checks when an uncurried function does not.

Gap to practical implementations of LTI. The restrictions that we discussed would be too severe in practice. In particular the hard-to-synthesize arguments problem, forbidding the use of unannotated lambda arguments, would prevent many uses of higher-order functions found in programs written

in languages like Java, Scala or TypeScript. Unsurprisingly, those languages do not adopt the same approach as Pierce and Turner. Instead, these languages address many of the problems above by using the information of instantiated arguments as soon as possible. Type inference proceeds from *left-to-right* and, as soon as instantiations are found, they are used to instantiate the remaining portion of the type. For example, the twice example above is accepted in all those programming languages, since once inference processes the argument 1 the instantiation for α is discovered and then we can check the lambda with the type $\mathsf{Int} \to \mathsf{Int}$. Note that this left-to-right approach will reject twice $(\lambda \mathsf{x}.\ \mathsf{x})\ 1$ for a flipped version of twice : $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$. Languages like Scala and F_c still reject such examples, which require a right-to-left inference approach. Thus, left-to-right inference can be seen as a pragmatic compromise to enable type checking many practical examples. Finally, note also that we need more flexibility than rule S-App-InfSpec as we need to flexibly choose between inference and checking per argument. The solution that we adopt in F_c allows for this flexibility and closely models what practical implementations of LTI do in practice.

In summary LTI has a complex specification, which makes it difficult to understand and introduces restrictions that are too severe in practice. Follow-up works on LTI have either reused the original specification of uncurried application rules [Odersky et al. 2001] and suffer from similar problems, or moved away from subtyping and developed a new specification with a complex logic for application dispatching [Jenkins and Stump 2018]. Practical implementations of LTI have significantly different designs that address some of the problems, but no clear specifications exist for these designs.

2.3 Background: Contextual Typing

Since our goal is to reformulate LTI using contextual typing, we first provide some background. The original work on contextual typing [Xue and Oliveira 2024] introduced Quantitative Type Assignment Systems (QTASs) as a declarative specification for contextual typing. A QTAS decorates a typing judgment with a *counter* that quantifies available type information. The concept of counter provides an accurate analogy in simple type systems such as the STLC. However, in more complex type systems, such as System F or systems with subtyping, counters do more than just quantifying existing type information. Thus, we change the terminology in this paper and use the term *mask* instead of counter to convey a more fine-grained notion that, not only quantifies available type information, but more generally pinpoints what parts of a type that are *known* or *unknown* from the context. The alternative perspective provided by masks is closely related to the notion of colors in colored type inference [Odersky et al. 2001]. Thus, we also employ colors to complement and aid in our presentation of the declarative type system. To be consistent with the change from counters to masks, we also change the terminology from QTAS to Contextual Type Assignment System (CTAS). Even though the terminology and syntax of masks changes with respect to the original work, it is still isomorphic to the original notion of counters, and there is no fundamental conceptual change on the original idea. The change is merely on presentation and terminology.

Contextual Type Assignment Systems (CTASs). Bidirectional typing takes an all-or-nothing approach, forbidding the propagation of partial type information. Contextual typing [Xue and Oliveira 2024] addresses this issue, and generalizes bidirectional typing. A contextual type system can be specified with a CTAS, which is a variant of a type assignment system, but decorated with *masks*, which indicate the availability of contextual information to type-check a term. Masks are constructed from *atomic masks*. An atomic mask can be either an opaque box ■ or a transparent box □. An opaque box denotes that the type information is not available, whereas a transparent box denotes available type information. Masks can be just atomic masks or application masks: $a\ m$, where a indicates the *partial* information known from arguments. ■ m means that one argument's type information is unavailable in the context, and □ m means that one argument's type is available.

$\boxed{\Gamma \vdash_m e : A}$ (Under environment Γ, expression e has type A with m contextual information.)

$$\frac{}{\Gamma \vdash_{\blacksquare} i : \mathsf{Int}}\ \mathsf{DLit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_{\blacksquare} x : A}\ \mathsf{DVar} \qquad \frac{\Gamma \vdash_{\square} e : A}{\Gamma \vdash_{\blacksquare} (e : A) : A}\ \mathsf{DAnn} \qquad \frac{\Gamma, x : A \vdash_{m-} e : B}{\Gamma \vdash_m \lambda x.\, e : A \to B}\ \mathsf{DLam}$$

$$\frac{\Gamma \vdash_{(\overline{a}\ m)} e_1 : A \to B \quad \Gamma \vdash_a e_2 : A}{\Gamma \vdash_m e_1\ e_2 : B}\ \mathsf{DApp} \qquad \frac{\Gamma \vdash_{\blacksquare} e : A \quad \Gamma \vdash_m A = B}{\Gamma \vdash_m e : B}\ \mathsf{DSub}$$

Fig. 1. Declarative typing rules for STLC.

In general masks where all information is available (such as $\square$ or $\square\square$) or no information is available (such as $\blacksquare$ or $\blacksquare\blacksquare$) can be modeled, respectively, in a traditional bidirectional type system with the inference and checking modes. In contrast, mixed masks, such as $\square\blacksquare$ or $\blacksquare\square$, where only some type information is available, have no direct correspondence to bidirectional typing. We present the CTAS for STLC in Fig. 1 to illustrate the key ideas. The syntax of contextual STLC is:

Types	$A, B, C, D ::= \mathsf{Int} \mid A \to B$
Typing Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$
Expressions	$e ::= i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e : A$
Atomic Masks	$a ::= \square \mid \blacksquare$
Masks	$m ::= a \mid a\ m$

To help understand the typing rules, we colorize the types based on their masks. Red types correspond to $\blacksquare$ masks and blue types to $\square$ masks. In other words, red types are unavailable in the context and are inferred by the term itself, while blue types are known from the context. We wish to remark that colors play no role in the theory but are used only for readability. Rules DLit, DVar, and DAnn apply to terms whose types can be inferred directly from their syntax, and the mask is $\blacksquare$ since no contextual information is required. Rule DLam states that a lambda term has type $A \to B$ if its mask can be decreased ($\square- = \square$ and $(\square\ m)- = m$), which means that A is known from the context. Rule DSub clears the context if the term e is inferable and the mask m matches type A. Besides changing the notion of counters to masks, there is one minor adaptation from the original type system [Xue and Oliveira 2024] to align with the presentation in Sec. 3. We use the notation $\Gamma \vdash_m A = B$ in the DSub rule. For STLC, subsumption is trivial, and it just amounts to syntactic equality and checking whether the shape of types matches masks.

The application rule is the most interesting. First we know that the argument e_2 is typeable with some mask a, The function part e_1 is then aware of the increased information by appending the flipped mask $\overline{a}$ to m ($\overline{\square} = \blacksquare$ and $\overline{\blacksquare} = \square$). The standard bidirectional rule for applications, using the inferred type of the function to check the arguments, will become a specialized instance of DApp when a is $\square$ and m is $\blacksquare$. In contrast, when a is $\blacksquare$ the type of the argument must be inferred, and this information can be used in e_1 as available contextual information. This enables the inference of lambdas as applicands and the encoding of let-binders. For example, $(\lambda x.\ \lambda y.\ x + y)\ 1\ 2$ is accepted:

$$\dfrac{\dfrac{}{\Gamma \vdash_{\blacksquare} 2 : \mathsf{Int}}\ \mathsf{DLit} \qquad \dfrac{\dfrac{}{\Gamma \vdash_{\blacksquare} 1 : \mathsf{Int}}\ \mathsf{DLit} \qquad \dfrac{\cdots}{\Gamma \vdash_{\square\square\blacksquare} (\lambda x.\ \lambda y.\ x + y) : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}}\ \mathsf{DLam}}{\Gamma \vdash_{\square\blacksquare} (\lambda x.\ \lambda y.\ x + y)\ 1 : \mathsf{Int} \to \mathsf{Int}}\ \mathsf{DApp}}{\Gamma \vdash_{\blacksquare} (\lambda x.\ \lambda y.\ x + y)\ 1\ 2 : \mathsf{Int}}\ \mathsf{DApp}$$

A naive implementation of DApp would rely on backtracking, since a choice between inference and checking would be needed for each argument. Instead, contextual typing provides a non-backtracking syntax-directed algorithm that can be efficiently implemented. We omit the presentation of the algorithm here, and discuss it instead in Sec. 5.

$\boxed{\Gamma \vdash_m e : A}$ (Under Γ, expression e has type A with m contextual information.)

$$\frac{}{\Gamma \vdash_{\blacksquare} i : \mathsf{Int}}\ \text{DT-Lit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_{\blacksquare} x : A}\ \text{DT-Var} \qquad \frac{\Gamma \vdash_{\square} e : A}{\Gamma \vdash_{\blacksquare} (e : A) : A}\ \text{DT-Ann} \qquad \frac{\Gamma, x : A \vdash_{m-} e : B}{\Gamma \vdash_m \lambda x.\, e : A \to B}\ \text{DT-Lam}$$

$$\frac{\Gamma \vdash_{(\overline{a}\ m)} e_1 : A \to B \qquad \Gamma \vdash_a e_2 : A}{\Gamma \vdash_m e_1\ e_2 : B}\ \text{DT-App} \qquad \frac{\Gamma \vdash_{\blacksquare} e : A \qquad \Gamma \vdash_m A \leq B}{\Gamma \vdash_m e : B}\ \text{DT-Sub}$$

$$\frac{\Gamma, \alpha \vdash_a e : A}{\Gamma \vdash_a \Lambda\alpha.\, e : \forall\alpha.\, A}\ \text{DT-TLam} \qquad \frac{\Gamma \vdash_{\blacksquare} e : \forall\alpha.\, B \qquad \Gamma \vdash_m [A/\alpha]\, B \leq C}{\Gamma \vdash_m e\ @A : C}\ \text{DT-TApp}$$

Fig. 2. Declarative typing rules.

3 Contextual System F

This section presents a CTAS specification for F_c, which provides a variant of System F with local (contextual) type inference. Notably implicit polymorphism is added in a completely modular way, without touching the application rule in Fig. 1. Instantiation is handled by contextual subsumption. The main change is in subtyping, which includes a suitably restricted instantiation rule.

3.1 Syntax and Typing

Syntax. Compared to the STLC CTAS in Sec. 2.3, we extend types and terms with standard System F constructs: type variables α, universal types $\forall\alpha.\, A$, type abstractions $\Lambda\alpha.\, e$, and type applications $e\ @A$. The design of masks stays the same in F_c.

Types	$A, B, C, D ::= \mathsf{Int} \mid A \to B \mid \alpha \mid \forall\alpha.\, A$
Expressions	$e ::= i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e : A \mid \Lambda\alpha.\, e \mid e\ @A$
Environments	$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha$

Typing. Fig. 2 shows the typing rules. Most rules are the same as those presented in Sec. 2.3. In DT-Sub, we have a subtyping $\Gamma \vdash_n A \leq B$ instead of $\Gamma \vdash_n A = B$. The subsumption rule is the entry point to implicit instantiation, where universal types can be instantiated into function types. The two extra rules in gray deal with two new constructs for System F. DT-TLam rule type-checks a type abstraction $\Lambda\alpha.\, e$ against a universal type $\forall\alpha.\, A$. With a contextual information available, it then type-checks its body e in the extended environment with type variable α. For type applications $e\ @A$, DT-TApp first infers the type of e, which is expected to be a universal type $\forall\alpha.\, B$. Then, it checks that the instantiated type $[A/\alpha]\, B$ is a subtype of type C by leveraging m contextual information. The type C is the expected type of the whole type application expression.

Example. We show the example of inferring a type for id id 1 below, only for the typing portion of the derivation. In this example Γ is $\mathsf{id} : \forall\alpha.\, \alpha \to \alpha$.

$$\dfrac{\dfrac{\dfrac{\dots}{\vdash_{\square\square\blacksquare} \mathsf{id} : (\forall\alpha.\, \alpha \to \alpha) \to \mathsf{Int} \to \mathsf{Int}}\ \text{DT-Sub} \quad \dfrac{\dots}{\vdash_{\blacksquare} \mathsf{id} : \forall\alpha.\, \alpha \to \alpha}\ \text{DT-Var}}{\Gamma \vdash_{\square\blacksquare} \mathsf{id}\ \mathsf{id} : \mathsf{Int} \to \mathsf{Int}}\ \text{DT-App} \quad \dfrac{}{\Gamma \vdash_{\blacksquare} 1 : \mathsf{Int}}\ \text{DT-Lit}}{\Gamma \vdash_{\blacksquare} \mathsf{id}\ \mathsf{id}\ 1 : \mathsf{Int}}\ \text{DT-App}$$

Two key points arise in this derivation. First, DT-App uses the argument types to guide the inference of the type of the function [Xie and Oliveira 2018]. We infer the types of two arguments id and 1, and then propagate the type information to the context of the function id. This use of DT-App is the opposite of the conventional rule for bidirectional typing where we infer the type of the function and then use the input type to check the arguments instead. Second, the subsumption

$\boxed{\Gamma \vdash_m A \leq B}$ (Under Γ, type A is a subtype of B and m contextual information is known for B.)

$$\frac{}{\Gamma \vdash_{\blacksquare} A \leq A}\ \text{DS-Refl} \qquad \frac{}{\Gamma \vdash_{\square} \mathsf{Int} \leq \mathsf{Int}}\ \text{DS-Int} \qquad \frac{}{\Gamma \vdash_{\square} \alpha \leq \alpha}\ \text{DS-Var} \qquad \frac{\Gamma, \alpha \vdash_{\square} A \leq B}{\Gamma \vdash_{\square} \forall \alpha.\, A \leq \forall \alpha.\, B}\ \text{DS-}\forall$$

$$\frac{\Gamma \vdash_{\square} C \leq A \qquad \Gamma \vdash_{m \dot{-}} B \leq D}{\Gamma \vdash_m A \to B \leq C \to D}\ \text{DS-Arr} \qquad \frac{\Gamma \vdash B \qquad A^{\alpha}_{(a\ m)} \qquad \Gamma \vdash_{(a\ m)} [B/\alpha]A \leq C \to D}{\Gamma \vdash_{(a\ m)} \forall \alpha.\, A \leq C \to D}\ \text{DS-}\forall\text{L}$$

$$A^{\alpha}_{\blacksquare} = \alpha \notin \mathsf{FV}(A) \quad (1) \qquad (A \to B)^{\alpha}_{(\square\ m)} = \alpha \in \mathsf{FV}(A) \quad (4)$$

$$A^{\alpha}_{\square} = \mathsf{true} \quad (2) \qquad (A \to B)^{\alpha}_{(a\ m)} = \alpha \notin \mathsf{FV}(A) \wedge B^{\alpha}_{m} \quad (5)$$

$$\alpha^{\alpha}_{\square} = \mathsf{true} \quad (3) \qquad (\forall \beta.\, A)^{\alpha}_{(a\ m)} = A^{\alpha}_{(a\ m)} \quad (6)$$

Fig. 3. Declarative subtyping rules and instantiability.

rule relates the type of id ($\forall \alpha.\ \alpha \to \alpha$) to its instantiated type (($\forall \alpha.\ \alpha \to \alpha$) $\to$ Int $\to$ Int). The subtyping derivation and instantiation details are discussed in the next.

3.2 Contextual Subtyping

The contextual subtyping rules are shown in Fig. 3. The mask indicates how much contextual information is available for the supertype. The mask $\square$ means we have full information about the supertype, leading to traditional subtyping. Rules DS-Int, DS-Var and DS-$\forall$ all use the $\square$ mask. Because we are modeling System F, all subtyping checks with a $\square$ mask imply syntactic equivalence checks. The $\square$ m mask means that the first input type of the supertype is known to be available in the context, and it can be used for instantiating polymorphic functions and performing subtyping checks. For function types (rule DS-Arr), we decrease the mask ((a m)$\dot{-}$ = m and $\square\dot{-} = \square$), and do contravariant checks of input types and covariant checks of output types.

Instantiation rule. The DS-$\forall$L rule is key to the implicit instantiation of polymorphic types. A polymorphic type is a subtype of a function type if its type argument α appears in known parts of contextual information, characterized as instantiability. We guess the type B, and then check that the substituted result $[B/\alpha]A$ is a subtype of function type $C \to D$. Keen readers may notice that the mask is $a\ m$ in rule DS-$\forall$L. This means that we have arguments in the context, and instantiation can only happen in function application. Moreover, the mask $a\ m$ explains why the supertype is a function type: since we are in the context of an application, we expect the supertype to be a function type. Therefore, the formulation of rule DS-$\forall$L nicely expresses the observation about local type inference that implicit instantiations can *only be triggered by applications*.

Instantiability. The instantiability restriction A^{α}_{m} that appears in rule DS-$\forall$L determines when solutions for α can be found. It can be read as "α is instantiable in type A with m contextual information". We present its rules in Fig. 3. Our instantiability judgment uses the mask as a condition to indicate the known parts of types. Essentially instantiability covers three cases:

- Type variable α does not appear in type A. In this case, there is no need to find a solution for the instantiation; for example, consider $\forall \alpha.$ Int $\to$ Int. This case can be handled by (1) and (2).
- Type variable α appears in known output types. For example, in f : $\forall \alpha.$ Int $\to \alpha \to \alpha \vdash$ (f 1) : Int $\to$ Int, the mask for f is $\square\square$, indicating that $\alpha \to \alpha$ is in the known part of the type. By further checking the occurrence of α in $\alpha \to \alpha$ we can guess the solution for α, handled by (2). Rule (3) deals with the case where a variable α covers multiple known positions. For example, in f : $\forall \alpha.\ \alpha \vdash$ (f 1) : Int, we know both the input and output types of f, from arguments and annotations, we can instantiate α with Int $\to$ Int to accept this example. We allow this by defining

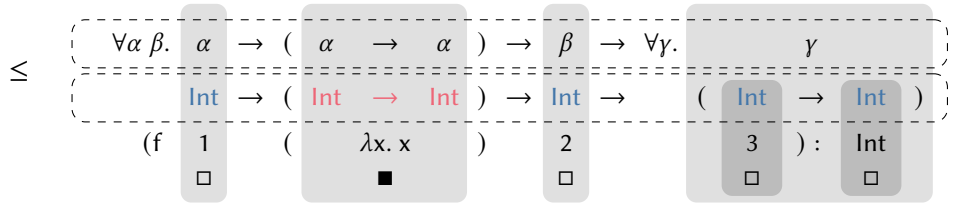

Fig. 4. Visualization of instantiability, subtyping and arguments.

the relation of isomorphic □, represented as $\widetilde{\square}$ ($\widetilde{\square} := \square \mid \square\,\widetilde{\square}$). Once the mask is isomorphic to □ and corresponds to a variable α, we are able to guess the α.

- Type variable α appears in the position of known type arguments. The simplest case is id 1, and for this case the mask is □■, indicating the input type α is known, handled by (4).

(5) and (6) simply cover the inductive cases and delegate instantiability to the subparts of the type. From (5) we can see our instantiability enforces a left-to-right order and exploits the known information from the input to the output types. This helps it to align with the algorithm in Sec. 5.

Correspondence between masks, types, and arguments. Fig. 4 illustrates the correspondence between masks, subtypes, supertypes, and arguments in subtyping and instantiability, using the slightly artificial example (f 1 (λx. x) 2 3) : Int. The first row shows the type bound to variable f, which is $\forall\alpha\ \beta.\ \alpha \to (\alpha \to \alpha) \to \beta \to \forall\gamma.\ \gamma$. The second row shows f's instantiated types when it is applied to its arguments and annotated. Another way to read the first two rows (surrounded in dashed boxes) is that they represent the subtype and supertypes of a subtyping statement, with the mask □■□□□ shown in the fourth row. The third row shows the application expression, with f's arguments and outer annotation Int. We highlight their correspondence with vertical gray blobs. In the type of f, α and β are instantiable based on the two □ masks, which are provided by the inferable arguments 1 and 2. The argument λx. x can be checked against $\alpha \to \alpha$ with full contextual information, thanks to the instantiability of α. Finally, the instantiability of γ is determined by the mask □□, according to rule (3), using combined information from the inferable argument 3 and the annotation Int. This example demonstrates how F_c can handle problematic examples for Pierce and Turner's LTI approach, such as those shown in Sec. 2.2.

Properties. Subtyping is reflexive for the ■ mask as shown by rule DS-Refl, and we show that reflexivity also holds for the □ mask. Furthermore, subtyping is transitive for arbitrary masks.

Theorem 3.1 (Reflexivity of Subtyping). $\Gamma \vdash_{\square} A \le A$.

Theorem 3.2 (Transitivity of subtyping). *If* $\Gamma \vdash_m A \le B$ *and* $\Gamma \vdash_m B \le C$, *then* $\Gamma \vdash_m A \le C$.

Examples. We complete our example discussed in Sec. 3.1 with the subtyping derivation. In this example, we first apply the DS-∀L rule because the subtype is polymorphic and the supertype is a function type. We know that α is instantiable since it appears in the input type indicated by the □ mask. Then, we guess the solution for α, perform the substitution, and obtain a function type as the subtype. DS-Arr then eliminates the mask and checks the input and output types. Subsequently, we encounter a similar situation and need to apply the DS-∀L rule again. We guess α to have type Int and perform the remaining subtyping checks.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \quad \Gamma \vdash_{\blacksquare} \mathsf{Int} \le \mathsf{Int}}\ \text{DS-Refl}}{\cdots \quad \Gamma \vdash_{\square\blacksquare} [\mathsf{Int}/\alpha]\alpha \to \alpha \le \mathsf{Int} \to \mathsf{Int}}\ \text{DS-Arr}}{\cdots \quad \Gamma \vdash_{\square\blacksquare} \forall\alpha.\ \alpha \to \alpha \le \mathsf{Int} \to \mathsf{Int}}\ \text{DS-∀L}}{\cdots \quad \Gamma \vdash_{\square\square\blacksquare} [\forall\alpha.\ \alpha \to \alpha/\alpha]\alpha \to \alpha \le (\forall\alpha.\ \alpha \to \alpha) \to \mathsf{Int} \to \mathsf{Int}}\ \text{DS-Arr}}{\Gamma \vdash_{\square\square\blacksquare} \forall\alpha.\ \alpha \to \alpha \le (\forall\alpha.\alpha \to \alpha) \to \mathsf{Int} \to \mathsf{Int}}\ \text{DS-∀L}$$

$\boxed{\Gamma \vdash e : A \rightsquigarrow e'}$ (Under environment Γ, expression e has type A and elaborates to e'.)

$$\frac{}{\Gamma \vdash i : \mathsf{Int} \rightsquigarrow i}\ \textsf{IF-Lit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow x}\ \textsf{IF-Var} \qquad \frac{\Gamma, x : A \vdash e : B \rightsquigarrow e'}{\Gamma \vdash \lambda x.\, e : A \to B \rightsquigarrow \lambda x.\, e'}\ \textsf{IF-Lam}$$

$$\frac{\Gamma \vdash e_1 : A \rightsquigarrow e_1' \qquad \Gamma \vdash A \triangleright B \to C \qquad \Gamma \vdash e_2 : B \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : C \rightsquigarrow e_1'\ (e_2' : B)}\ \textsf{IF-App} \qquad \frac{\Gamma, \alpha \vdash e : A \rightsquigarrow e'}{\Gamma \vdash e : \forall \alpha.\, A \rightsquigarrow \Lambda \alpha.\, (e' : A)}\ \textsf{IF-}\forall$$

Fig. 5. Elaboration rules for implicit system F.

3.3 Relation to Implicit System F

We relate F_c with variants of implicit System F in the literature. Implicit System F [Chrząszcz 1998; Mitchell 1988] is a variant of System F that does not require any annotations or explicit type applications. It supports impredicative polymorphism and can type-check all the examples we present in the previous sections. For soundness we relate F_c to a variant of System F [Chrząszcz 1998] that adopts the unrestricted $\forall$L rule for polymorphic subtyping presented in Sec. 1.

Soundness. We show that F_c is sound with respect to implicit System F after type erasure. This result essentially shows that our restricted instantiation rule models a subset of the programs allowed by the unrestricted rule $\forall$L.

Theorem 3.3 (Soundness to Implicit System F). *If* $\Gamma \vdash_m e : A$, *then* $\Gamma \vdash \mathsf{erase}(e) : A$.

Completeness and annotatability. Implicit System F with polymorphic subtyping can handle deep instantiation; however, our implicit instantiation is shallow (restricting instantiation to rank-1 quantifiers) and is triggered solely by applications. To establish a completeness result, we create a tailored implicit System F for our setting: we remove polymorphic subtyping and the subsumption rule, and embed the logic of instantiation into the application rules.

We present the elaboration rules of the tailored implicit system F (F_i) in Fig. 5. The elaboration from F_i to F_c is straightforward by inserting annotations into appropriate places. IF-Lit and IF-Var are standard type assignment rules and elaborate to identical terms. IF-Lam is typed with a function type and elaborates to $\lambda x.\, e'$, where e' comes from the elaborated result of its body. IF-App is particularly interesting: we do not expect the e_1 to have a function type, instead, we have an instantiation judgment to transform a type into a function type:

$$\frac{}{\Gamma \vdash A \to B \triangleright A \to B}\ \textsf{Inst-Base} \qquad \frac{\Gamma \vdash B \qquad \Gamma \vdash [B/\alpha]A \triangleright C \to D}{\Gamma \vdash \forall \alpha.\, A \triangleright C \to D}\ \textsf{Inst-}\forall$$

Inst-Base is the base case that takes a function type and returns the same result; and Inst-$\forall$ indicates that if we have a polymorphic type $\forall \alpha.\, A$ and can guess the type B, then we can substitute α in A with B and obtain $C \to D$ as the instantiation result of the polymorphic type. Instantiation is simple but powerful. For example, it can instantiate the type $\forall \alpha.\, \alpha \to \alpha$ to $\mathsf{Int} \to \mathsf{Int}$. More importantly, it allows empty types to be instantiated with any type, such as instantiating $\forall \alpha.\, \alpha$ to $\mathsf{Int} \to \mathsf{Int}$. Having the instantiation in the application reflects the characteristics of F_c, where only applications can trigger implicit instantiations. Note that we always annotate the argument in the elaboration of applications. To avoid the burden of excessive annotations, it is possible to adopt a syntactic check in the transformation stage, to drop annotations on obviously inferable arguments, characterized as generic consumers in Sec. 5. IF-$\forall$ is the introduction rule for polymorphic types, and it allows an expression to be typed with a type $\forall \alpha.\, A$, provided that it can be typed with A under the extended environment. We note that we insert the annotation on the elaborated term e', in case we get unintended elaborated results such as $(\Lambda \alpha.\, \lambda \mathrm{x.\ x})\ 1$, which is untypeable in F_c.

We present the completeness (annotatability) theorem below. In principle, to type-check a F_i term in F_c, it is sufficient to place annotations only on arguments and bodies of type abstractions.

Theorem 3.4 (Annotatability of F_i). *If* $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_{\square} e' : A$.

We can derive a corollary and obtain an inferable term by adding a top-level annotation.

Corollary 3.5 (Annotatability of F_i). *If* $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_{\blacksquare} (e' : A) : A$.

4 Matching Subtyping

This section introduces an intermediate subtyping relation, called *matching subtyping*, which serves as a bridge between declarative and algorithmic subtyping. In the algorithmic formulation of F_c, subtyping is where local constraint solving happens and where the main algorithmic challenge lies. Matching subtyping *specifies* a key algorithmic idea: the use of *matching variables* to track solved instantiations. Matching subtyping is sound and complete to the declarative subtyping in Sec. 3.2.

4.1 Matching Variables

A key point emphasized by Pierce and Turner [2000] is that local type inference avoids long-distance constraints such as unification variables. Instead of solving a unification problem, local type inference, as well as contextual type inference, requires solving only a *matching* [Bürckert 1989; Bürckert 1986; Stirling 2009] (also known as one-sided unification) problem. Before introducing matching subtyping, we first explain the difference between matching and unification variables.

Unification variables. Many (global) type inference algorithms employ *unification* variables to keep track of information found during inference. For example, when inferring the type for $\lambda x.\ x + 1$, they may create a unification variable $\hat{\alpha}$ as a placeholder for the type of x and then discover that $\hat{\alpha} = \mathsf{Int}$ from analyzing the body of the lambda abstraction. Unification variables are powerful, and they even allow solving problems such as $\lambda f.\ (f\ 1) + 2$. Here f would be given a placeholder type $\hat{\alpha}$. When encountering the application $f\ 1$, two new unification variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$ are created and $\hat{\alpha}$ is solved to $\hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2$. Then $\hat{\alpha}_1$ and $\hat{\alpha}_2$ can be solved to Int. As this example shows, solutions for unification variables can themselves contain other unification variables.

Matching variables. Matching variables can be seen as a restricted form of unification variables where solutions to a matching variable *cannot* contain other matching variables. Thus, with matching variables, $\hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2$ would not be allowed, since the solution $\hat{\alpha}_1 \to \hat{\alpha}_2$ contains other (matching) variables. From this discussion we can see that matching variables would be quite restrictive for doing inference of anonymous functions, which partly explains why local type inference approaches typically do not provide such feature. The primary application of matching in local type inference techniques and F_c is to infer type instantiations for polymorphic functions.

4.2 Syntax and Auxiliary Definitions

Matching subtyping shares the same type syntax as declarative subtyping. However, its type variables serve two distinct purposes: (1) universal type variables, and (2) matching type variables representing solutions to instantiation. While most work distinguishes these variables using different syntax and separate constructors, we differentiate them based on their environment entries, similar to the representation used by Jiang et al. [2025]. Matching type variables can only be created during subtyping and do not leak into typing. In other words, solving instantiations is a *local* process: if solutions cannot be found with only the local information then the subtyping statement is rejected. The new subtyping environment has the syntax:

Subtyping Environments $\quad \Delta ::= \cdot \mid \Delta, \alpha \mid \Delta, \hat{\alpha} = A$

Subtyping environments introduce a new entry $\hat{\alpha} = A$ for matching variables, to represent solutions found for instantiations. This approach aligns with our Agda formalization, where we use de Bruijn indices to represent type variables, with their meaning determined by the corresponding entries. Moreover, it avoids substitutions in algorithmic subtyping, as we will explain in detail in Sec. 5.

Ground types and ground environments. A type is ground ($\Gamma \vDash A$ and $\Gamma \shortmid \Delta \vDash A$) if it contains only universal type variables. In typing, all types are ground, since typing can only use universal variables. Well-formed environments ($\vdash \Gamma$ and $\vdash \Gamma \shortmid \Delta$) contain only ground types. In other words, matching entries in the environments cannot have solutions that contain matching variables.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \Gamma \vDash A}{\vdash \Gamma, x : A} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \alpha} \qquad \frac{\vdash \Gamma}{\vdash \Gamma \shortmid \cdot} \qquad \frac{\vdash \Gamma \shortmid \Delta}{\vdash \Gamma \shortmid \Delta, \alpha} \qquad \frac{\vdash \Gamma \shortmid \Delta \qquad \Gamma \shortmid \Delta \vDash A}{\vdash \Gamma \shortmid \Delta, \hat{\alpha} = A}$$

Polarity. Polarity determines which type (subtype or supertype) in subtyping must contain only ground types. We employ the following syntax to capture polarity:

$$\text{Polarity} \qquad \leq^{\pm} ::= \leq^{+} \mid \leq^{-}$$

Matching employs two subtyping modes: positive ($\leq^{+}$) and negative ($\leq^{-}$), where $\leq^{\pm}$ serves as a meta-variable denoting either mode. In the positive mode ($\leq^{+}$), supertypes must be ground, while subtypes may contain matching variables that refer to solutions. The negative mode ($\leq^{-}$) enforces the opposite constraint. In contravariant cases, where we swap the positions of types in the subtyping relation, the polarity is reversed accordingly.

Grounding. Matching subtyping employs a grounding operation ($[\Delta]A$) that transforms types into ground types by replacing variables with solutions recorded in the environment, shown below. This relation takes an environment and a type as input and produces a ground type as output. This operation is also the critical component to relate the intermediate system to the declarative system, and is used in soundness and completeness statements.

$$[\Delta]\mathsf{Int} = \mathsf{Int} \qquad [\Delta]\alpha = A \quad (\text{if } \hat{\alpha} = A \in \Delta) \qquad [\Delta]\alpha = \alpha \quad (\text{otherwise})$$

$$[\Delta](A \to B) = [\Delta]A \to [\Delta]B \qquad [\Delta](\forall \alpha.\, A) = \forall \alpha.\, [\Delta, \alpha]A$$

4.3 Subtyping

Matching subtyping is shown in Fig. 6 and has the judgment form $\Gamma \shortmid \Delta \vdash_m A \leq^{\pm} B$. Different from declarative subtyping, matching subtyping takes two environments. Typing environments (Γ) are never modified in the subtyping derivation, but are required since types may contain type variables (universal variables) in the typing environment. Subtyping environments (Δ) are where solutions to instantiation problems are recorded, and are also used to track universal variables local to subtyping. Another difference is that matching subtyping is *polarized* by using $\leq^{\pm}$.

Rule IS-Refl uses a ■ mask in positive mode, establishing that a type A is a subtype of its grounded result. Intuitively, IS-Refl serves as a base case and appears at the leaves of the derivation tree, where we have run out of masks (contextual information) and it is time to produce an inference result. Rules IS-Int, IS-Var, IS-Arr, and IS-∀ resemble their counterparts in the declarative system and are in both modes, though in IS-Arr we must reverse the polarity in its contravariant subcase, indicated by $\leq^{\mp}$. IS-Arr-S is in positive mode and also deals with the function types. It covers the cases with □ masks and ■ masks. In practice, they represent two different situations where the type C is known from the inference result of the argument or to check the argument. For matching subtyping, we can merge these two rules into one and distinguish them in the algorithm in Sec. 5. IS-∀L is in positive mode. Unlike DS-∀L in the declarative system, we do not perform immediate substitution; instead, we record a solution $\hat{\alpha} = B$ in the subtyping environment. The correctness of the guessed solution is verified by the base cases for solved variables: the two rules at the bottom,

$\boxed{\Gamma \mid \Delta \vdash_m A \leq^+ B}$ (type A is a subtype of B and m contextual information is known for B)

$\boxed{\Gamma \mid \Delta \vdash_m A \leq^- B}$ (type A is a subtype of B and m contextual information is known for A)

$$\frac{}{\Gamma \mid \Delta \vdash_{\blacksquare} A \leq^+ [\Delta]A}\ \text{IS-Refl} \qquad \frac{}{\Gamma \mid \Delta \vdash_{\square} \mathsf{Int} \leq^{\pm} \mathsf{Int}}\ \text{IS-Int} \qquad \frac{\alpha \in (\Gamma \mid \Delta)}{\Gamma \mid \Delta \vdash_{\square} \alpha \leq^{\pm} \alpha}\ \text{IS-Var}$$

$$\frac{\Gamma \mid \Delta \vdash_{\square} C \leq^{\mp} A \qquad \Gamma \mid \Delta \vdash_{\square} B \leq^{\pm} D}{\Gamma \mid \Delta \vdash_{\square} A \to B \leq^{\pm} C \to D}\ \text{IS-Arr} \qquad \frac{\Gamma \mid \Delta \vdash_{\square} C \leq^{-} A \qquad \Gamma \mid \Delta \vdash_{m} B \leq^{+} D}{\Gamma \mid \Delta \vdash_{(a\ m)} A \to B \leq^{+} C \to D}\ \text{IS-Arr-S}$$

$$\frac{\Gamma \mid \Delta, \alpha \vdash_{\square} A \leq^{\pm} B}{\Gamma \mid \Delta \vdash_{\square} \forall \alpha.\, A \leq^{\pm} \forall \alpha.\, B}\ \text{IS-}\forall \qquad \frac{\Gamma \vdash B \qquad A^{\alpha}_{(a\ m)} \qquad \Gamma \mid \Delta, \hat{\alpha} = B \vdash_{(a\ m)} A \leq^{+} C \to D}{\Gamma \mid \Delta \vdash_{(a\ m)} \forall \alpha.\, A \leq^{+} C \to D}\ \text{IS-}\forall\text{L}$$

$$\frac{\Gamma \mid \Delta[\hat{\alpha} = A] \vdash_m A \leq^+ B}{\Gamma \mid \Delta[\hat{\alpha} = A] \vdash_m \alpha \leq^+ B}\ \text{IS-Var-L} \qquad \frac{}{\Gamma \mid \Delta[\hat{\alpha} = A] \vdash_{\square} A \leq^- \alpha}\ \text{IS-Var-R}$$

Fig. 6. Matching subtyping.

which are new to this system. Rule IS-Var-L is in positive mode. When we have a matching variable as a subtype, then we look up the environment and obtain the solution A (we use the syntactic sugar $\Delta[\hat{\alpha} = A]$ for $\Delta_1, \hat{\alpha} = A, \Delta_2$), then perform the subtyping between the solution and the supertype B. IS-Var-R is in negative mode, and can only occur with the $\square$ mask. If a matching variable occurs as a supertype, we find its solution and check the equivalence between the subtype and the solution.

Examples. We rerun our examples using matching subtyping rules. The key differences from the example shown in Sec. 3.2 are three: (1) In IS-∀L, instead of directly substituting the type with the guessed solution, we record it as a matching variable in the subtyping environment; (2) matching variables are retrieved when a variable appears as the subtype in IS-Var-L; (3) the interpretation of masks depends on the polarity (reflected in colors): if the subtyping is in negative mode, the mask will describe the contextual information about subtypes instead of supertypes.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta \vdash_{\square} \forall\beta.\, \beta \to \beta \leq^- \alpha}\ \text{IS-Var-R} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta, \hat{\beta} = \mathsf{Int} \vdash_{\blacksquare} \beta \leq^+ \mathsf{Int}}\ \text{IS-Refl}}{\cdots \quad \Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta, \hat{\beta} = \mathsf{Int} \vdash_{\square\blacksquare} \beta \to \beta \leq^+ \mathsf{Int} \to \mathsf{Int}}\ \text{IS-Arr-S}}{\cdots \quad \Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta \vdash_{\square\blacksquare} \forall\beta.\, \beta \to \beta \leq^+ \mathsf{Int} \to \mathsf{Int}}\ \text{IS-}\forall\text{L}}{\Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta \vdash_{\square} \forall\beta.\, \beta \to \beta \leq^- \alpha \qquad \Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta \vdash_{\square\blacksquare} \alpha \leq^+ \mathsf{Int} \to \mathsf{Int}}\ \text{IS-Var-L}}{\cdots \quad \Gamma \mid \hat{\alpha} = \forall\beta.\, \beta \to \beta \vdash_{\square\square\blacksquare} \alpha \to \alpha \leq^+ (\forall\beta.\, \beta \to \beta) \to \mathsf{Int} \to \mathsf{Int}}\ \text{IS-Arr-S}}{\Gamma \mid \cdot \vdash_{\square\square\blacksquare} \forall\alpha.\, \alpha \to \alpha \leq^+ (\forall\beta.\, \beta \to \beta) \to \mathsf{Int} \to \mathsf{Int}}\ \text{IS-}\forall\text{L}$$

4.4 Soundness and Completeness

To show that matching subtyping is equivalent to declarative subtyping, we prove soundness and completeness results. We distinguish between the declarative ($\vdash^{\mathrm{d}}_m$) and matching ($\vdash^{\mathrm{i}}_m$) formulation of subtyping in the theorems below by using superscript letters "d" and "i", respectively. The key idea for establishing the soundness is to eliminate variables in subtyping environments: universal variables in Δ are merged to the typing environment Γ (denoted as $\Gamma \ll \Delta$ and defined below) and matching variables are removed after grounding operations on types being performed.

$$\Gamma \ll \cdot = \Gamma \qquad \Gamma \ll (\Delta, \alpha) = (\Gamma \ll \Delta), \alpha \qquad \Gamma \ll (\Delta, \hat{\alpha} = A) = \Gamma \ll \Delta$$

Theorem 4.1 (Soundness of Subtyping). *If* $\Gamma \mid \Delta \vdash^{\mathrm{i}}_m A \leq^{\pm} B$, *then* $(\Gamma \ll \Delta) \vdash^{\mathrm{d}}_m [\Delta]A \leq [\Delta]B$.

Completeness can be stated by constructing a subtyping environment Δ, and replacing relevant parts of types with matching variables based on the polarity.

Theorem 4.2 (Completeness of Subtyping).
- *If* $\Gamma \vdash^{\mathrm{d}}_{m} [\Delta]A \leq B$, *then* $\Gamma \shortmid \Delta \vdash^{\mathrm{i}}_{m} A \leq^{+} B$;
- *If* $\Gamma \vdash^{\mathrm{d}}_{m} A \leq [\Delta]B$, *then* $\Gamma \shortmid \Delta \vdash^{\mathrm{i}}_{m} A \leq^{-} B$.

Corollary 4.3 (Completeness and Soundness of Subtyping). *For ground types A and B,*
- *If* $\Gamma \shortmid \cdot \vdash^{\mathrm{i}}_{m} A \leq^{\pm} B$, *then* $\Gamma \vdash^{\mathrm{d}}_{m} A \leq B$
- *If* $\Gamma \vdash^{\mathrm{d}}_{m} A \leq B$, *then* $\Gamma \shortmid \cdot \vdash^{\mathrm{i}}_{m} A \leq^{+} B$.

5 Algorithmic Type Inference

In this section, we introduce an algorithmic system and show its equivalence to the CTAS of F_c via matching subtyping. To realize the algorithm, we adopt several techniques. (1) Following the design of contextual typing [Xue and Oliveira 2024], we use a *teleportation-based* algorithm where type-checking tasks for arguments are deferred by pushing them to the *surrounding contexts*, and then dealt with when encountering their *application consumers*. (2) *Matching variables with and without solutions*: Unlike what IS-∀L rule does in matching subtyping, where a solution of the quantifier is immediately guessed, in algorithmic subtyping, we use matching variables that are unsolved until a candidate solution is found during subtyping. (3) *Input and output environments*: Like other type inference algorithms [Bosman et al. 2023; Dunfield and Krishnaswami 2013; Mercer et al. 2022], we employ input and output environments to track the solving of matching variables.

5.1 Syntax

The algorithmic system shares the same syntax for terms, types and typing environments. Like matching subtyping, environments are separated into typing environments and subtyping environments. The difference is the addition of a new entry for *unsolved* matching variables $\hat{\alpha}$ in subtyping environments, and two more syntactic categories: surrounding contexts and generic consumers. Surrounding contexts describe the contextual information available for the expression. Contexts can be either empty ($\blacksquare$), a full type A and terms e followed by another surrounding context. For example, when inferring the type of expression id id 1, the surrounding context of id is $\boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare$, indicating id is applied to a term id and then applied to another term 1. *Generic consumers* classify terms that are inferable and can be generically dealt with contextual subsumption.

Subtyping Environments	$\Delta ::= \cdot \mid \Delta, \alpha \mid \Delta, \hat{\alpha} \mid \Delta, \hat{\alpha} = A$
Surrounding Contexts	$\Sigma ::= \blacksquare \mid A \mid \boxed{e} \rightsquigarrow \Sigma$
Generic Consumers	$g ::= i \mid x \mid e : A \mid \Lambda\alpha.\, e$

5.2 Typing

The complete set of typing rules is shown in Fig. 7. Most typing rules follow the design of the original algorithm for contextual typing. Only subsumption rule AT-Sub is modified to accommodate our new subtyping rules, and three additional rules are added to type-check type abstractions and applications. In AT-Sub, we first infer the type of the generic consumer g, and then check the inferred type A with the context Σ. We call subtyping with an initial empty subtyping environment. $\Gamma \vdash A \leq^{+} \Sigma \rightsquigarrow B$ is a syntactic sugar for $\Gamma \shortmid \cdot \vdash A \leq^{+} \Sigma \dashv \cdot \rightsquigarrow B$. Algorithmic subtyping computes the expected inference result B, which is an instantiation of A. For example, A can be a polymorphic type $\forall\alpha.\, \alpha \to \alpha$, and B can be its instantiation $\mathsf{Int} \to \mathsf{Int}$ if the context is $\boxed{1} \rightsquigarrow \blacksquare$. AT-TLam1 infers the type of the type abstraction if its body infers the type A in an extended environment and AT-TLam2 checks the type abstraction with the type $\forall\alpha.\, B$ if the body can be checked by the type B in the extended environment. We note that AT-TLam2 overlaps with the subsumption rule when the context is a polymorphic type $\forall\alpha.\, A$, since $\Lambda\alpha.\, e$ is treated as a generic consumer. This overlapping is not harmful since, in this situation, the rule AT-TLam2 will subsume the subsumption rule, and one can always prioritize AT-TLam2 over the subsumption rule in the implementation.

$\boxed{\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A}$ (Under typing environment Γ and context Σ, term e infers the type A.)

$$\frac{}{\Gamma \vdash \blacksquare \Rightarrow i \Rightarrow \mathsf{Int}}\ \text{AT-Lit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash \blacksquare \Rightarrow x \Rightarrow A}\ \text{AT-Var} \qquad \frac{\Gamma \vdash A \Rightarrow e \Rightarrow B}{\Gamma \vdash \blacksquare \Rightarrow e : A \Rightarrow A}\ \text{AT-Ann}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow e \Rightarrow C}{\Gamma \vdash A \to B \Rightarrow \lambda x.\, e \Rightarrow A \to C}\ \text{AT-Lam1} \qquad \frac{\Gamma \vdash \blacksquare \Rightarrow e_2 \Rightarrow A \qquad \Gamma, x : A \vdash \Sigma \Rightarrow e \Rightarrow B}{\Gamma \vdash \boxed{e_2} \rightsquigarrow \Sigma \Rightarrow \lambda x.\, e \Rightarrow A \to B}\ \text{AT-Lam2}$$

$$\text{AT-TLam1}\ \frac{\Gamma, \alpha \vdash \blacksquare \Rightarrow e \Rightarrow A}{\Gamma \vdash \blacksquare \Rightarrow \Lambda\alpha.\, e \Rightarrow \forall\alpha.\, A} \qquad \text{AT-TLam2}\ \frac{\Gamma, \alpha \vdash B \Rightarrow e \Rightarrow A}{\Gamma \vdash \forall\alpha.\, B \Rightarrow \Lambda\alpha.\, e \Rightarrow \forall\alpha.\, A}$$

$$\text{AT-App}\ \frac{\Gamma \vdash \boxed{e_2} \rightsquigarrow \Sigma \Rightarrow e_1 \Rightarrow A \to B}{\Gamma \vdash \Sigma \Rightarrow e_1\ e_2 \Rightarrow B}$$

$$\text{AT-TApp}\ \frac{\Gamma \vdash \blacksquare \Rightarrow e \Rightarrow \forall\alpha.\, B \qquad \Gamma \vdash [A/\alpha]B \leq^{+} \Sigma \rightsquigarrow C}{\Gamma \vdash \Sigma \Rightarrow e\ @A \Rightarrow C} \qquad \text{AT-Sub}\ \frac{\Gamma \vdash \blacksquare \Rightarrow g \Rightarrow A \quad \Sigma \neq \blacksquare \quad \Gamma \vdash A \leq^{+} \Sigma \rightsquigarrow B}{\Gamma \vdash \Sigma \Rightarrow g \Rightarrow B}$$

Invariants: (1) $\vdash \Gamma$ (2) $\Gamma \vDash A$

Fig. 7. Typing rules for the algorithmic system.

For type applications, we first infer the type of term e to a polymorphic type $\forall\alpha.\, B$, and then check the instantiated type $[A/\alpha]B$ with the context Σ. The inference result C of the subtyping is the result for the type application. Algorithmic typing has two invariants: (1) all typing environments are well-formed; (2) all inferred types are ground types.

Examples. We show a derivation for (id id) 1 below. Suppose Γ is $\mathsf{id} : \forall\alpha.\, \alpha \to \alpha$. We first eliminate the applications until we reach the generic application consumer id. We have a context $\boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare$, and then use this context as the supertype for the type of id. Our algorithmic subtyping will compute the inference result $(\forall\alpha.\, \alpha \to \alpha) \to \mathsf{Int} \to \mathsf{Int}$, which is the type of id.

$$\dfrac{\dfrac{\dfrac{\ldots \quad \Gamma \shortmid \cdot \vdash \forall\alpha.\, \alpha \to \alpha \leq^{+} \boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare \dashv \cdot \rightsquigarrow (\forall\alpha.\, \alpha \to \alpha) \to (\mathsf{Int} \to \mathsf{Int})}{\Gamma \vdash \boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare \Rightarrow \mathsf{id} \Rightarrow (\forall\alpha.\, \alpha \to \alpha) \to (\mathsf{Int} \to \mathsf{Int})}\ \text{AT-Sub}}{\Gamma \vdash \boxed{1} \rightsquigarrow \blacksquare \Rightarrow \mathsf{id}\ \mathsf{id} \Rightarrow \mathsf{Int} \to \mathsf{Int}}\ \text{AT-App}}{\Gamma \vdash \blacksquare \Rightarrow (\mathsf{id}\ \mathsf{id})\ 1 \Rightarrow \mathsf{Int}}\ \text{AT-App}$$

5.3 Subtyping

Algorithmic subtyping distinguishes two kinds of judgments: *subtyping inference* (Fig. 8) and *subtyping checking* (Fig. 9). Subtyping inference has the form $\Gamma \shortmid \Delta \vdash A \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow B$ whereas subtyping checking is of the form $\Gamma \shortmid \Delta \vdash A \leq^{\pm} B \dashv \Delta'$. Both judgments take typing (Γ) and subtyping (Δ) environments as inputs and yield an output subtyping environment (Δ'). In subtyping inference, a type is compared against a surrounding context and produces a supertype B based on the surrounding context. In subtyping checking, a type is compared directly against a supertype B. Furthermore, subtyping inference always operates in positive polarity (surrounding contexts come from typing and are ground) and is interdependent with typing, while subtyping checking can operate in both polarities and is independent of other judgments.

Subtyping inference has several important invariants, shown at the bottom of Fig. 8. Firstly, all environments are ground environments (1). Secondly, the output environment has the same length and order as the input environment (2). The only difference is that unsolved matching variables in input environments may be solved in output environments. This invariant is formalized in Sec. 5.4

$\boxed{\Gamma \vdash \Delta \vdash A \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow B}$ (Under Γ and Δ, type A is the subtype of Σ and outputs Δ' and B.)

$$\frac{\Gamma \vdash \Delta \vdash_{\text{closed}} A}{\Gamma \vdash \Delta \vdash A \leq^{+} \blacksquare \dashv \Delta \rightsquigarrow [\Delta]A}\ \text{AS-Empty} \qquad \frac{\Gamma \vdash \Delta \vdash A \leq^{+} B \dashv \Delta'}{\Gamma \vdash \Delta \vdash A \leq^{+} B \dashv \Delta' \rightsquigarrow B}\ \text{AS-Type} \qquad \frac{\Gamma \vdash \Delta, \hat{\alpha} \vdash A \leq^{+} \boxed{e} \rightsquigarrow \Sigma \dashv \Delta', \hat{\alpha} =^{?} C \rightsquigarrow B}{\Gamma \vdash \Delta \vdash \forall \alpha.\, A \leq^{+} \boxed{e} \rightsquigarrow \Sigma \dashv \Delta' \rightsquigarrow B}\ \text{AS-}\forall\text{L}$$

$$\frac{\Gamma \vdash \Delta \vdash_{\text{closed}} A \qquad \Gamma \vdash [\Delta]A \Rightarrow e \Rightarrow A' \qquad \Gamma \vdash \Delta \vdash B \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow D}{\Gamma \vdash \Delta \vdash A \to B \leq^{+} \boxed{e} \rightsquigarrow \Sigma \dashv \Delta' \rightsquigarrow [\Delta]A \to D}\ \text{AS-Trm-C}$$

$$\frac{\Gamma \vdash \Delta \vdash_{\text{open}} A \quad \Gamma \vdash \blacksquare \Rightarrow e \Rightarrow C \qquad \Gamma \vdash \Delta \vdash C \leq^{-} A \dashv \Delta' \quad \Gamma \vdash \Delta' \vdash B \leq^{+} \Sigma \dashv \Delta'' \rightsquigarrow D}{\Gamma \vdash \Delta \vdash A \to B \leq^{+} \boxed{e} \rightsquigarrow \Sigma \dashv \Delta'' \rightsquigarrow C \to D}\ \text{AS-Trm-O}$$

$$\frac{\Gamma \vdash \Delta[\hat{\alpha} = A] \vdash A \leq^{+} \Sigma \dashv \Delta[\hat{\alpha} = A] \rightsquigarrow B}{\Gamma \vdash \Delta[\hat{\alpha} = A] \vdash \alpha \leq^{+} \Sigma \dashv \Delta[\hat{\alpha} = A] \rightsquigarrow B}\ \text{AS-SVar} \qquad \frac{}{\Gamma \vdash A \Longrightarrow A}\ \text{AS-Infer-Type}$$

$$\frac{\Gamma \vdash \boxed{e} \rightsquigarrow \Sigma \Longrightarrow A}{\Gamma \vdash \Delta[\hat{\alpha}] \vdash \alpha \leq^{+} \boxed{e} \rightsquigarrow \Sigma \dashv \Delta[\hat{\alpha} = A] \rightsquigarrow A}\ \text{As-Infs} \qquad \frac{\Gamma \vdash \blacksquare \Rightarrow e \Rightarrow A \qquad \Gamma \vdash \Sigma \Longrightarrow B}{\Gamma \vdash \boxed{e} \rightsquigarrow \Sigma \Longrightarrow A \to B}\ \text{AS-Infer-Con}$$

Invariants: (1) $\vdash \Gamma \vdash \Delta$ and $\vdash \Gamma \vdash \Delta'$ (2) $\Delta \subseteq \Delta'$ (3) $\Gamma \vdash \Delta \vDash \Sigma$ (4) $\Gamma \vdash \Delta \vDash B$ (5) $\Gamma \vdash \Delta' \vdash_{\text{closed}} A$

Fig. 8. Rules for subtyping inference.

as environment extension (Fig. 11). Thirdly, polarity in subtyping checking implies the ground property: in the positive mode, the supertype is ground, whereas in the negative mode the subtype is ground (3). Finally, the inference result of subtyping inference is always ground (4).

An important distinction that plays a fundamental role in the subtyping rules is between *closed* and *open* types. A closed type is a type where all matching variables have already been solved, whereas an open type may contain unsolved variables. For instance consider the type $\alpha \to \alpha$. If the subtyping environment contains an unsolved variable for α (for instance, $\Delta \triangleq \hat{\alpha}$) then the type is said to be open. In contrast, if the subtyping environment contains a solution (for instance, $\Delta \triangleq (\hat{\alpha} = \mathsf{Int})$) then the type is said to be closed. All types are closed in output environments (5).

Subtyping inference rules. The rules for subtyping inference are presented in Fig. 8. In rule AS-Empty, when the context is empty, and A is a closed type, we use the input environment Δ as output, and compute the grounded result of A as the inference result. Rule AS-Type applies when the context is a ground type B. This case corresponds to traditional subtyping, and so we just switch to subtyping checking. The inferred result is just B. In rule AS-$\forall$L a universal type $\forall\alpha.\, A$, matches a term context $\boxed{e} \rightsquigarrow \Sigma$. In the algorithm we simply introduce an unsolved matching variable $\hat{\alpha}$ as a placeholder for the solution found later. Note that it is possible that no solution is found. For instance, in the case that we have $\forall\alpha.\, \mathsf{Int} \to \mathsf{Int}$, and the argument is 1. The rule works in both cases: whether a solution is found or not and the notation $\hat{\alpha} =^{?} C$ expresses these two possibilities (either we get a solved or an unsolved variable). Rule AS-SVar applies when the subtype is a solved matching variable. In this case we look up the solution in the environment and compare it to the context. The last rule AS-Infs applies when the subtype is an unsolved matching variable α. The only possibility that we have to find a solution in this case is that we must have inferable term arguments, and we must also have an output type (the context ending with a full type). The auxiliary judgment $\Gamma \vdash \Sigma \Longrightarrow A$ takes a context with inferable arguments and an output type and computes a type A that becomes the solution to $\hat{\alpha}$.

Left-to-right inference. AS-Trm-C and AS-Trm-O are two interesting rules: the subtype is a function type $A \to B$, and we have a term context $\boxed{e} \rightsquigarrow \Sigma$ in the place of supertype. Although

$\boxed{\Gamma \mid \Delta \vdash A \leq^{\pm} B \dashv \Delta'}$ (Under Γ and Δ, A is a subtype of B and computes Δ'.)

$$\frac{}{\Gamma \mid \Delta \vdash \mathsf{Int} \leq^{\pm} \mathsf{Int} \dashv \Delta}\ \text{AS-Int} \qquad \frac{\alpha \in (\Gamma \mid \Delta)}{\Gamma \mid \Delta \vdash \alpha \leq^{\pm} \alpha \dashv \Delta}\ \text{AS-UVar}$$

$$\frac{}{\Gamma \mid \Delta[\hat{\alpha}] \vdash \alpha \leq^{+} A \dashv \Delta[\hat{\alpha} = A]}\ \text{AS-MVar-L} \qquad \frac{}{\Gamma \mid \Delta[\hat{\alpha}] \vdash A \leq^{-} \alpha \dashv \Delta[\hat{\alpha} = A]}\ \text{AS-MVar-R}$$

$$\frac{}{\Gamma \mid \Delta[\hat{\alpha} = A] \vdash \alpha \leq^{+} A \dashv \Delta[\hat{\alpha} = A]}\ \text{AS-SVar-L} \qquad \frac{}{\Gamma \mid \Delta[\hat{\alpha} = A] \vdash A \leq^{-} \alpha \dashv \Delta[\hat{\alpha} = A]}\ \text{AS-SVar-R}$$

$$\frac{\Gamma \mid \Delta \vdash C \leq^{\mp} A \dashv \Delta' \qquad \Gamma \mid \Delta' \vdash B \leq^{\pm} D \dashv \Delta''}{\Gamma \mid \Delta \vdash A \to B \leq^{\pm} C \to D \dashv \Delta''}\ \text{AS-Arr} \qquad \frac{\Gamma \mid \Delta, \alpha \vdash A \leq^{\pm} B \dashv \Delta', \alpha}{\Gamma \mid \Delta \vdash \forall \alpha.\, A \leq^{\pm} \forall \alpha.\, B \dashv \Delta'}\ \text{AS-}\forall$$

Invariants ($\leq^{+}$): (1) $\vdash \Gamma \mid \Delta$ and $\vdash \Gamma \mid \Delta'$ (2) $\Delta \subseteq \Delta'$ (3) $\Gamma \mid \Delta \vDash B$ (4) $\Gamma \mid \Delta' \vdash_{\text{closed}} A$
Invariants ($\leq^{-}$): (1) $\vdash \Gamma \mid \Delta$ and $\vdash \Gamma \mid \Delta'$ (2) $\Delta \subseteq \Delta'$ (3) $\Gamma \mid \Delta \vDash A$ (4) $\Gamma \mid \Delta' \vdash_{\text{closed}} B$

Fig. 9. Rules for subtyping checking.

their syntax is overlapped, they can be distinguished by the two disjoint conditions to have a deterministic algorithm. We first check whether A is closed or open. If it is closed, we can safely use its grounded result to check the argument e using typing. If A is open, we infer the type of e, and obtain the type C. C will be used for subtyping checking with A and computes an environment Δ' which contains the solution of matching variables in A. Δ' will be used as the input environment for the subtyping inference in the final premise. The consequence of using open and closed types to distinguish whether we perform inference or checking on the argument is that we have a left-to-right bias. For example, when A is $\alpha \to \alpha$ and e is λx. x where α is unsolved in the environments, we will immediately reject this case even though the solution for α can be found later. In both rules the final premise simply checks whether the output type B matches with the remaining context Σ.

Subtyping checking rules. We present the subtyping checking rules in Fig. 9. AS-Int and AS-UVar are simply equality checks and can be in both modes, producing the same output environment. AS-MVar-L is in positive mode, which tells us that A is a ground type, thus it can be used as a solution for unsolved variable α. Rule AS-MVar-R is a dual rule for the negative polarity. AS-SVar-L and AS-SVar-R cover the case where the subtype is a solved matching variable α. In this case we just have to check that the already found solution is the same as the newly found solution. AS-Arr is for the subtyping rule for arrow types and it is mostly standard. The only notable point is that we have to reverse the polarity for subtyping checking of the input types. Rule AS-$\forall$ covers universal types in a standard way: we just have to check whether the bodies of the universal type are in a subtyping relation in an extended environment with the universal variable α.

Examples. We continue our example shown in Sec. 5.2. We use Γ as id : $\forall\alpha.\ \alpha \to \alpha$, Δ as $\hat{\alpha} = \forall\beta.\ \beta \to \beta, \mathbb{I}$ as Int and omit inference and checking for terms in contexts for space reasons. First we have the polymorphic type as the subtype and the term context as the supertype, we will create an unsolved matching variable α, which will be later solved as $\forall\beta.\ \beta \to \beta$ from the inference result of id. Then we will have $\forall\beta.\ \beta \to \beta$ compared with the remaining context $\boxed{1} \rightsquigarrow \blacksquare$. We create a new unsolved matching variable $\hat{\beta}$, which will be solved as Int in the output environment.

5.4 Invariants and Properties

In this part we show some invariants and key properties including the decidability of the algorithm.

Environment extension. Environment extension captures the information growth of the algorithm. As subtyping proceeds, the algorithm may discover more solutions to unsolved variables. Thus

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \shortmid \hat{\alpha} \vdash \forall\beta.\ \beta \to \beta \leq^{-} \alpha \dashv \Delta}\ \text{AS-EVar-R} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \shortmid \Delta, \hat{\beta} \vdash \mathbb{I} \leq^{-} \beta \dashv \Delta, \hat{\beta} = \mathbb{I}}\ \text{AS-Empty} \quad \cdots}{\Gamma \shortmid \Delta, \hat{\beta} \vdash \beta \to \beta \leq^{+} \boxed{1} \rightsquigarrow \blacksquare \dashv \Delta, \hat{\beta} = \mathbb{I} \rightsquigarrow \mathbb{I} \to \mathbb{I}}\ \text{AS-Trm-O}}{\Gamma \shortmid \Delta \vdash \forall\beta.\ \beta \to \beta \leq^{+} \boxed{1} \rightsquigarrow \blacksquare \dashv \Delta \rightsquigarrow \mathbb{I} \to \mathbb{I}}\ \text{AS-}\forall\text{L}}{\Gamma \shortmid \Delta \vdash \alpha \leq^{+} \boxed{1} \rightsquigarrow \blacksquare \dashv \Delta \rightsquigarrow \mathbb{I} \to \mathbb{I}}\ \text{AS-SVar}}{\Gamma \shortmid \hat{\alpha} \vdash \alpha \to \alpha \leq^{+} \boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare \dashv \Delta \rightsquigarrow (\forall\alpha.\ \alpha \to \alpha) \to (\mathbb{I} \to \mathbb{I})}\ \text{AS-Trm-O}}{\Gamma \shortmid \cdot \vdash \forall\alpha.\ \alpha \to \alpha \leq^{+} \boxed{\mathsf{id}} \rightsquigarrow \boxed{1} \rightsquigarrow \blacksquare \dashv \cdot \rightsquigarrow (\forall\alpha.\ \alpha \to \alpha) \to (\mathbb{I} \to \mathbb{I})}\ \text{AS-}\forall\text{L}$$

Fig. 10. Subtyping inference derivation for (id id) 1.

$$\frac{}{\Delta, \alpha, \Delta' \subset_{\alpha} \Delta, \alpha, \Delta'}\ \text{EV-UVar} \qquad \frac{}{\Delta, \hat{\alpha}, \Delta' \subset_{\alpha} \Delta, \hat{\alpha} = A, \Delta'}\ \text{EV-EVar} \qquad \frac{}{\Delta, \hat{\alpha} = A, \Delta' \subset_{\alpha} \Delta, \hat{\alpha} = A, \Delta'}\ \text{EV-SVar}$$

$$\frac{}{\Delta \subseteq_{\mathtt{Int}} \Delta}\ \text{E-Int} \qquad \frac{\Delta \subset_{\alpha} \Delta'}{\Delta \subseteq_{\alpha} \Delta'}\ \text{E-Var} \qquad \frac{\Delta \subseteq_{A} \Delta' \qquad \Delta' \subseteq_{B} \Delta''}{\Delta \subseteq_{A \to B} \Delta''}\ \text{E-Arr} \qquad \frac{\Delta, \alpha \subseteq_{A} \Delta', \alpha}{\Delta \subseteq_{\forall\alpha.\ A} \Delta'}\ \text{E-}\forall$$

Fig. 11. Environment Extension under variables and types.

the output environment always has at least as many solved variables as the corresponding input environment. Environment extension appears in several forms in our proof, but here we only formally show two main versions, informally explaining the remainder when they arise. Interested readers can find the details in the Agda proof in the supplementary material. We present the environment extension under types and under type variables in Fig. 11. In $\Delta \subseteq \Delta'$, Δ represents the input environment and Δ' represents the output environment in subtyping. Due to matching subtyping, we are able to enforce a strict invariant for this relation: the environments must have the same length, and the entries must remain in the same order. Thus, the only difference between input and output environments is that the unsolved matching variables in the input environment may be solved in the output environment. $\Delta \subseteq_A \Delta'$ requires that any free variable in A, if appearing as an unsolved matching variable, must be solved in the output environment Δ', leaving other entries unchanged. This relation relies on the auxiliary relation $\Delta \subset_a \Delta'$, indicating that only the entry corresponding to a will be affected between the two environments. Interesting lemmas include:

Lemma 5.1 (Subtyping Environment Extension).

- *If* $\Gamma \shortmid \Delta \vdash A \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow B$, *then* $\Delta \subseteq_A \Delta'$.
- *If* $\Gamma \shortmid \Delta \vdash A \leq^{+} B \dashv \Delta'$, *then* $\Delta \subseteq_A \Delta'$; • *If* $\Gamma \shortmid \Delta \vdash A \leq^{-} B \dashv \Delta'$, *then* $\Delta \subseteq_B \Delta'$.

This lemma shows that all the free matching variables in type A are solved in the output environment Δ' for the two subtyping judgements.

Decidability. Our algorithm is decidable by the following two theorems.

Theorem 5.2 (Decidability of Typing and subtyping).

- $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$ *is decidable.* • $\Gamma \shortmid \Delta \vdash A \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow B$ *is decidable.*

The decidability of subtyping and typing are proved simultaneously. The measure used for typing is $|\Sigma| + |e|$, and the measure for subtyping is $(|\Sigma|, |A|^{\Delta}, |A|^{\forall})$. Below is the definition of sizes: $|A|^{\Delta}$ counts solved matching variables in A, and $|A|^{\forall}$ counts universal quantifiers in A.

$$\begin{array}{lll}
|\blacksquare| = 0 & |i|, |x| = 1 & |a|^{\Delta[a=A]} = 1 \\
|A| = 1 & |\lambda x.\ e| = |e| + 1 & |A \to B|^{\Delta} = |A|^{\Delta} + |B|^{\Delta} \\
|\boxed{e} \rightsquigarrow \Sigma| = |e| + |\Sigma| + 2 & |e\ @A|, |e : A| = |e| + 2 & |\forall\alpha.\ A|^{\Delta} = |A|^{\Delta,\alpha} \\
 & |e_1\ e_2| = |e_1| + |e_2| + 3 & |A|^{\Delta} = 0 (\text{otherwise})
\end{array}$$

5.5 Soundness and Completeness via Matching Subtyping

To prove the equivalence between a CTAS and our algorithmic system, we must address two main gaps between the two systems: masks and contexts, and environments.

Soundness. Constructing surrounding contexts from masks presents a challenging task. Readers might assume that surrounding contexts provide more information than masks, but this is not the case. Masks precisely indicate how arguments type-check in the typing: inference or checking, while contexts do not provide this information. In previous work [Xue and Oliveira 2024], soundness was proved by reconstructing the original application from the arguments in surrounding contexts and demonstrating that the resulting program is well-typed in CTAS. The previous proof also relied on a customized induction principle, making it difficult to extend those systems.

In this work, we simplify the proof strategy by introducing a new variant of the algorithmic system: $\Gamma \vdash \Delta \vdash A \leq^+ \Sigma \dashv \Delta' \rightsquigarrow B \downarrow m$, whose rules can be found in the appendix. Different from the algorithmic formulation, this variant computes an extra output: the mask m to describe how contexts are used during inference. This algorithmic variant is shown to be sound with respect to matching subtyping. For environments, we reuse the output environments Δ' as the subtyping environments in the matching subtyping, while keeping their typing environment Γ the same.

Lemma 5.3 (Soundness of Instantiation). *If* $\Gamma \vdash \Delta[\hat{\alpha}] \vdash A \leq^+ \Sigma \dashv \Delta'[\hat{\alpha} = B] \rightsquigarrow C \downarrow m$, *then* A^{α}_{m}.

This lemma states that each matching variable in type A is solved in the algorithm, which can be captured by the instantiability relation A^{α}_{m} used in matching subtyping.

Theorem 5.4 (Generalized Soundness of Subtyping). *If* $\Gamma \vdash \Delta \vdash A \leq^+ \Sigma \dashv \Delta' \rightsquigarrow B \downarrow m$, *then* $\Gamma \vdash \Delta' \vdash_m A \leq^+ B$.

Theorem 5.5 (Generalized Soundness of Typing). *If* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A \downarrow m$, *then* $\Gamma \vdash_m e : A$.

By further establishing the equivalence between the algorithm and its variant, we can derive the soundness of the original algorithmic system as corollaries.

Corollary 5.6 (Soundness of Typing).

- *If* $\Gamma \vdash \blacksquare \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash_{\blacksquare} e : A$;
- *If* $\Gamma \vdash B \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash_{\square} e : B$.

Completeness. Building contexts from masks is straightforward, since masks provide sufficient information to create expressions that are inferable or checkable. Following the proof strategy of Xue and Oliveira, we define a relation $\Gamma \vdash \Delta \vdash (m, B) \sim \Sigma$ that creates a context Σ with the mask m and the type B. The main challenges in establishing completeness statements are threefold.

(1) We have to align intermediate and algorithmic environments. In matching subtyping, environments contain solved matching variables only, and the solutions are obtained immediately in the IS-∀L rule. In contrast, the algorithm uses input environments with unsolved matching variables, introduced by the AS-∀L rule, and solutions are discovered incrementally as the types are analyzed. For example, in the matching subtyping statement $\Gamma \vdash \cdot \vdash_{\square\square\blacksquare} \forall \alpha\ \beta.\ \alpha \to \beta \to \beta \leq^+ \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$, the solutions for α and β are already present in the environment, whereas the algorithm determines these solutions only in processing the types α and $\beta \to \beta$.

(2) We have to deal with the gap between different treatments of arguments. In matching subtyping, application masks are eliminated without side conditions in the subtyping rules. For example, IS-Arr-S can be applied in any situation, whether the argument is handled by checking or inference. However, in the algorithm, to ensure determinism, we enforce several side conditions to control when to perform inference or checking. For example, to type-check the term succ 1, our declarative system can assign either a □ or ■ mask. However, the algorithm will only infer the type of arguments when the information is not sufficient for checking, following the open restriction. The consequence is that the argument 1 can only be checked in the algorithm.

(3) Matching subtyping may choose a less precise solution than the algorithm. For instance, if the subtype is $\forall\alpha.\ \alpha$, with mask $\square\square$ and supertype $\mathsf{Int} \to \mathsf{Int}$, matching subtyping might guess solutions like $\forall\beta.\ \beta \to \beta$ or $\mathsf{Int} \to \mathsf{Int}$. In contrast, the algorithm always computes a unique type. For example, with context $\boxed{1} \rightsquigarrow \mathsf{Int}$, the matching variable α is solved as $\mathsf{Int} \to \mathsf{Int}$. Thus, the solutions in environments in the two systems do not always coincide.

To address (1), we define a refined environment extension $\Delta \subseteq_{A_m} \Delta'$ to build the input environment, similar to the relation shown in Fig. 11. In the refined environment extension, the treatment of free variables in A depends on its position as indicated by the mask m. This relation is processed backwards and non-deterministically: we know the output environment Δ' from matching subtyping and need to create the input environment Δ, which contains the unsolved matching variables by altering the relevant entries in Δ'. This relation is delicate, as if we change the solved matching variables to unsolved matching variables in the input environments, and then build the algorithm derivation, we must ensure that the algorithm will successfully solve those variables. For example, if we have the matching subtyping $\Gamma \shortmid \hat{\alpha} = \mathsf{Int}, \hat{\beta} = \mathsf{Int} \vdash_{\square} \alpha \leq^{+} \mathsf{Int}$, we can create the input environment $\hat{\alpha}, \hat{\beta} = \mathsf{Int}$ or $\hat{\alpha} = \mathsf{Int}, \hat{\beta} = \mathsf{Int}$ but not $\hat{\alpha}, \hat{\beta}$ or $\hat{\alpha} = \mathsf{Int}, \hat{\beta}$. Furthermore, we prove several environment stability lemmas. These allow us to state that typing and subtyping hold even when the entries in the input and output environments are altered, provided that the free variables in the types remain unchanged. For example, $\Gamma \shortmid \hat{\alpha}, \hat{\beta} \vdash \alpha \leq^{+} \mathsf{Int} \vdash \hat{\alpha} = \mathsf{Int}, \hat{\beta}$, is interchangeable with $\Gamma \shortmid \hat{\alpha}, \hat{\beta} = \mathsf{Int} \vdash \alpha \leq^{+} \mathsf{Int} \vdash \hat{\alpha} = \mathsf{Int}, \hat{\beta} = \mathsf{Int}$, since $\hat{\beta}$ is irrelevant to this derivation.

For (2), we need to prove a general subsumption lemma for the algorithmic system to handle cases where checking is expected, but an inference result is obtained.

Lemma 5.7 (Typing implies Subtyping). *If* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash A \leq^{+} \Sigma \rightsquigarrow A$.

Corollary 5.8 (Subsumption of Algo. Typing). *If* $\Gamma \vdash \blacksquare \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash A \Rightarrow e \Rightarrow A$.

For (3), we detect that the scenario appears with empty-like types. For example, if the subtype is $\forall\alpha.\ \alpha$ or $\forall\alpha\ \beta.\ \alpha \to \beta$, there will be no constraints for guessing the solution of the variables that appear *only* at the end of the type. We perform a case analysis on this situation and construct a new output environment based on the information from the supertype of matching subtyping. We only show the simplified version of completeness here, with the full story detailed in the appendix.

Theorem 5.9 (Generalized Completeness of Subtyping). *If* $\Gamma \shortmid \Delta' \vdash_m A \leq^{+} B$, $\Delta \subseteq_{A_m} \Delta'$, *and* $\Gamma \shortmid \Delta \vdash (m, B) \sim \Sigma$, *then* $\Gamma \shortmid \Delta \vdash A \leq^{+} \Sigma \dashv \Delta' \rightsquigarrow B$.

Combined with completeness between declarative subtyping and matching subtyping in Thm. 4.2, we obtain completeness with respect to the declarative typing, with $m = \blacksquare$ and $\square$ cases as a corollary.

Theorem 5.10 (Generalized Completeness of Typing). *If* $\Gamma \vdash_m e : A$ *and* $\Gamma \shortmid \cdot \vdash (m, A) \sim \Sigma$, *then* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$.

Corollary 5.11 (Completeness of Typing).

- *If* $\Gamma \vdash_{\blacksquare} e : A$, *then* $\Gamma \vdash \blacksquare \Rightarrow e \Rightarrow A$;
- *If* $\Gamma \vdash_{\square} e : A$, *then* $\Gamma \vdash A \Rightarrow e \Rightarrow A$.

6 Extensions, Expressiveness Evaluation and Limitations

In this section, we discuss how F_c can be extended to support some common practical features. We also report on F_c's expressiveness evaluation through a comprehensive suite of examples by comparing our system with existing approaches in the literature and identify limitations. All features and examples in this section are implemented in our prototype.

6.1 Uncurried Functions

While our core calculus focuses on curried function applications, uncurried applications are equally important in practice. Supporting both curried and uncurried applications would provide better flexibility for programmers. We present a direct extension with dedicated syntax and rules below, and discuss an alternative approach using tuples in Sec. 6.2.

For syntax, we add uncurried function types $\overline{A} \to B$, uncurried lambda expressions $\lambda\overline{x}.\ e$, and uncurried applications $e(\overline{e})$, where the overline notation represents sequences of elements. The extension to the declarative system is straightforward, by mirroring the rules for curried functions and analyzing the argument list sequentially. The non-obvious rule for the algorithm is:

$$\frac{\overline{\Gamma \mid \Delta \vdash A_i \rightrightarrows e_i \rightrightarrows B_i \dashv \Delta'} \qquad \Gamma \mid \Delta' \vdash B \leq^{+} \Sigma \dashv \Delta'' \rightsquigarrow C}{\Gamma \mid \Delta \vdash \overline{A} \to B \leq^{+} \boxed{\overline{e}} \rightsquigarrow \Sigma \dashv \Delta'' \rightsquigarrow \overline{B} \to C}\ \text{AS-Term-UC}$$

When the subtype is an uncurried function type and the supertype is an uncurried argument context ($\boxed{\overline{e}} \rightsquigarrow \Sigma$), we need to analyze each type in $\overline{A}$ and then decide whether to infer the type of e_i or check it. This process is handled by the auxiliary judgment $\Gamma \mid \Delta \vdash A_i \rightrightarrows e_i \rightrightarrows B_i \dashv \Delta'$, whose logic follows the same strategy as how we handle curried function application in F_c. We first carry out case analysis on A_i to determine whether it is open or closed, and then perform the corresponding operation for inference or checking. Again, we note that the overline in AS-Term-UC denotes a sequence of such judgments, which follow a left-to-right order, passing the output environment to the next judgment. We present the full formalization in the appendix.

Our uncurried extension offers better locality compared to LTI by extending locality to cover the full set of arguments in an application, rather than being confined to individual (uncurried) applications. This enables us to accept examples such as $\mathsf{const}_3(1)(\mathsf{true})$, where LTI fails because it cannot solve type variables by considering arguments across different uncurried function calls.

6.2 Tuples

We extend F_c with tuples (e_1, e_2) and product types $A \times B$ by following the contextual typing recipe by Xue and Oliveira [2024] to allow contextual information propagation into tuple expressions beyond projections. For CTAS, we extend the mask with a projection mask: $m ::= \cdots \mid p\ m$ where p can be fst or snd to represent the context of projections. We show three key typing rules below. Like applications, projections fst e increase the contextual information of e, as shown in DT-Fst. For tuple expressions (e_1, e_2), if the mask is a (DT-Pair), we type-check both components with a; if the mask is a projection mask fst m (DT-Pair-Fst), we type-check the first component with mask m, and the second with $\blacksquare$. For the algorithmic system, we also have two new context entries fst $\rightsquigarrow \Sigma$ and snd $\rightsquigarrow \Sigma$, with rules that simply mirror the CTAS rules. We present the complete set of rules in the appendix, including typing and subtyping rules for both CTAS and the algorithmic system.

$$\text{DT-Pair}\ \frac{\Gamma \vdash_a e_1 : A \quad \Gamma \vdash_a e_2 : B}{\Gamma \vdash_a (e_1, e_2) : A \times B} \qquad \text{DT-Fst}\ \frac{\Gamma \vdash_{(\text{fst } m)} e : A \times B}{\Gamma \vdash_m \text{fst } e : A} \qquad \text{DT-Pair-Fst}\ \frac{\Gamma \vdash_m e_1 : A \qquad \Gamma \vdash_{\blacksquare} e_2 : B}{\Gamma \vdash_{(\text{fst } m)} (e_1, e_2) : A \times B}$$

Uncurrying via tupling. An alternative approach to modeling uncurried functions is to use tuples. Our design enables us to accept examples like $(\text{fst } (\lambda x.\ x, 2)) : \mathsf{Int} \to \mathsf{Int}$ and $(\text{fst } (\mathsf{id}, 1))\ 2$. However, there are challenges in propagating the type information across the tuple components. For example, suppose we have twice′ with the type $\forall\alpha.\ (\alpha, \alpha \to \alpha) \to \alpha$, F_c will reject the examples twice′ $(1, \lambda x.\ x)$ and twice′ $((\lambda y.\ (1, \lambda x.\ x))\ 2)$. The limitation stems from the fact that the current contextual information for arguments is limited to either all ($\square$) or nothing ($\blacksquare$). This is reflected in algorithmic subtyping as well: we either infer the type of the argument or have full information to check it. To support this, we need a more fine-grained mechanism for describing contextual

information about arguments. A possible option is to extend the atomic masks with (a_1, a_2) and design corresponding rules, but designing an equivalent algorithmic system would further require non-trivial changes. Thus, we leave this for future work.

6.3 Annotated Lambdas

Supporting annotated lambdas $\lambda x : A.\ e$ requires directly adding two rules to CTAS. The typing rules for annotated lambdas differ from those for unannotated lambdas in two ways: (1) The DT-AnLam1 rule covers one additional case where the context can be empty. Since we have the information of the binder, we can directly type-check the body with the extended environment; (2) The DT-AnLam2 rule covers the case where the argument can be checked against the binder's type, whereas an unannotated lambda can only rely on an inferable argument in the context. The algorithmic typing rules simply mirror the CTAS rules, and are straightforward to implement.

$$\frac{\Gamma, x : A \vdash_a e : B}{\Gamma \vdash_a \lambda x : A.\ e : A \to B}\ \text{DT-AnLam1} \qquad \frac{\Gamma, x : A \vdash_m e : B}{\Gamma \vdash_{a\ m} \lambda x : A.\ e : A \to B}\ \text{DT-AnLam2}$$

6.4 Evaluating Expressiveness

To demonstrate the expressiveness of F_c and assess its practical applicability, we have conducted an evaluation on a comprehensive suite of example programs from Serrano et al. [2018]. A table summarizing all examples is included in the appendix, together with a comparison to other approaches to impredicative polymorphism [Botlan and Rémy 2003; Leijen 2008, 2009; Mercer et al. 2022; Parreaux et al. 2024; Serrano et al. 2020; Vytiniotis et al. 2008]. Notably all of the examples can be encoded in F_c. This is not surprising, since F_c has explicit type applications and therefore we can always resort to explicit type applications and extra annotations to type check programs. Of course the other concern is how much explicit type information or program rewriting is needed. Several cases require additional annotations or η-expansions. Some of these cases arise from different design choices between local and global type inference. In particular, like other local type inference approaches, we do not support *let generalization*, *inference of lambdas* or *inference of type abstractions* (for example writing $\lambda x.\ x$ instead of $\Lambda a.\ \lambda(x : a).\ x$ for defining a polymorphic identity function). Therefore programs using such features in Serrano et al.'s programs will require more verbosity in F_c. There are however, two limitations that are cumbersome in practice and would be interesting directions for improvement in future work.

- **Order-relevant instantiation.** Following the current practice, F_c uses a left-to-right instantiation order, requiring the type variable's first occurrence to be solvable. As discussed in Section 2.2, this means that examples such as twice $(\lambda \mathsf{x}.\ \mathsf{x})$ 1 are rejected. Some of the F_c translations from Serrano et al.'s programs are affected by this limitation. An interesting direction for future work would be to study order-irrelevant instantiation, which would be more natural and less surprising to programmers, and would remove an important limitation in existing LTI implementations.
- **Application-triggered instantiation only.** Like LTI, F_c only triggers instantiation in expressions that are applied to arguments. This behavior rejects the example map id: the type of map is $\forall \alpha\ \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]$, requires the first input to be a function type, whereas id provides only a polymorphic type, and instantiation does not occur on id. The typable version is $\Lambda \alpha.$ map (id $@\alpha$), enforcing that id is a function type through an explicit type application. Similarly, map head (single ids) (where ids : $[\forall \alpha.\ \alpha \to \alpha]$) is translated as map (head $@(\forall \alpha.\ \alpha \to \alpha)$) (single ids) by instantiating the argument via a type application. In practice, rewriting such programs can be cumbersome, and it would be desirable to allow implicit instantiation without explicit arguments in such cases.

7 Related Work

Local Type Inference. Local type inference [Pierce and Turner 2000] has been extensively compared with in Sec. 2.1. In summary, F_c provides a more disciplined declarative specification that does not explicitly distinguish uncurried applications. F_c also provides a formal characterization of order relevance that is commonly adopted by practical implementations adopting LTI. F_c targets standard System F, while Pierce and Turner have shown that LTI can also deal with extensions of System F that require other forms of non-structural subtyping. In particular, their variant of System F includes $\top$ and $\bot$ types (and a more advanced formulation with bounded quantification). In their formulation, examples like $\mathsf{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha$, $\mathsf{f}\ 1\ \mathsf{true}$ are accepted and the inferred type is $\top \to \top \to \top$. From the specification point-of-view it is straightforward to deal with non-structural subtyping in F_c. Our Agda formalization already includes a variant with a declarative specification as well as a formulation of matching subtyping that extends F_c with $\top$ and $\bot$ types. We already proved soundness and completeness between matching and declarative subtyping, and have essentially all the results corresponding to Sec. 3 and 4 mechanized. The extension is simple, and only adds two additional rules for $\top$ and $\bot$ types, and modifying the matching variable rules in Fig. 6.

$$\frac{}{\Gamma \vdash_{\square} A \leq \top}\ \textsf{DS-Top} \qquad \frac{}{\Gamma \vdash_{n} \bot \leq A}\ \textsf{DS-Bot} \qquad \frac{\Gamma \shortmid \Delta[\hat{\alpha} = A] \vdash_{\square} B \leq^{-} A}{\Gamma \shortmid \Delta[\hat{\alpha} = A] \vdash_{\square} B \leq^{-} \alpha}\ \textsf{IS-Var-R}$$

As adding these rules requires no changes to other rules, it demonstrates the modularity and scalability of F_c. However, the addition of $\top$ and $\bot$ further implies some changes to algorithms since an eager approach to finding instantiations cannot be adopted. Pierce and Turner show that the use of matching is also helpful in dealing with $\top$ and $\bot$ and a lazier approach to solving instantiations. We are confident that those ideas can be adapted to local contextual typing, but we have not yet formalized the algorithm, which is left for future work.

The majority of follow-up work to LTI adopts designs with uncurried applications, following Pierce and Turner's approach. As a result, many of the limitations discussed in Sec. 2.2 still apply. Colored local type inference (CLTI) [Odersky et al. 2001; Plociniczak 2016] directly extends Pierce and Turner [2000]. Our presentation of the CTAS borrows colors and ideas from CLTI. CLTI decorates types to be *inherited* (or known from the context) and *synthesized* (or unknown from the context) to enable partial type information (from arguments) to assist the inference of polymorphic functions. Unlike CLTI, instead of decorating types, we use masks and avoid changes in the type syntax. The original purpose of CLTI is different from contextual typing, and enables the inference of $\mathsf{g} : \forall\alpha.\ (\mathsf{Int} \to \alpha) \to \alpha \vdash \mathsf{g}\ (\lambda\mathsf{x}.\ \mathsf{x})$, by exploiting the known parts (Int) of the input type $\mathsf{Int} \to \alpha$ and argument $\lambda\mathsf{x}.\ \mathsf{x}$. This example is rejected by LTI and F_c. An extension of F_c allowing such examples seems possible, but we have not formalized it yet. Implicit polarized F [Mercer et al. 2022] reformulated System F in terms of call-by-push-value (CBPV) [Levy 2001]. Mercer et al. observe that the shift structure mediating between computations (functions) and values in CBPV coincides well with the common practice in type inference to utilize application spines (uncurried applications) to deal with fully unambiguous function calls. However, as acknowledged by Mercer et al., their work is not proposed as a practical type inference algorithm, since a CBPV calculus is usually used as an intermediate language that programmers do not directly interact with. To use it as the base system, programmers have to adopt an unconventional polarized term and type syntax.[2] F_c, on the other hand, can type-check programs written in traditional System F syntax.

The only other approach that adopts curried syntax like ours is spine-local type inference [Jenkins and Stump 2018], making it the closest to our approach in terms of the programs that are accepted. Spine-local has a hybrid approach where annotations are explored prior to arguments, but then

[2]This polarization has a different meaning from the one in F_c.

the remaining inference proceeds left-to-right. The approach in base F_c is purely left-to-right, thus annotations are explored at the end. The annotation still helps to find solutions to instantiate functions, but it does not provide more information when checking the arguments. This makes F_c reject some examples from spine-local type inference, such as:

$$\mathsf{pair} : \forall \alpha\ \beta.\ \alpha \to \beta \to \langle \alpha \times \beta \rangle \vdash \mathsf{pair}\ (\lambda \mathsf{x}.\ \mathsf{x})\ 1 \Leftarrow \langle (\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Int} \rangle$$

We have experimented with a variant of F_c that can accept programs like the above. We have proved all properties for this variant as well, which is included in our Agda formalization. All the rules are detailed in the appendix. To support curried application, spine-local type inference uses extra judgments to dispatch the application rules in the form of spines and use meta-functions to find application heads. Unlike F_c and all other works on LTI, spine-local type inference does not employ subtyping. This design works well for System F, but seems hard to extend with new features. For example, the addition of $\top$ and $\bot$ to the specification of spine-local type inference is not straightforward, since subtyping is needed. Furthermore, the addition of these types interacts with applications. For example, $\mathsf{f} : \bot \vdash \mathsf{f}\ 1 : \mathsf{Int}$ is a valid application in a system with $\bot$. Thus, their application judgments would require non-trivial modifications in the presence of subtyping. As we have seen earlier, our CTAS can incorporate features such as $\top$ and $\bot$ easily and modularly.

ML-style First-class Polymorphism. There is a long line of work on extending ML-style type inference with first-class polymorphism [Botlan and Rémy 2003; Garrigue and Rémy 1999; Serrano et al. 2018; Vytiniotis et al. 2006], which in general requires fewer annotations compared to F_c and LTI. HMF [Leijen 2008] aims at a conservative extension over Hindley-Milner and requires annotations only on polymorphic parameters and ambiguous impredicative instantiations. Its full version takes application spines as a whole into account, meaning it could have a different result for $f\ e_1\ e_2$ and $\mathsf{let}\ g = f\ e_1\ \mathsf{in}\ g\ e_2$. FreezeML [Emrich et al. 2020] introduces a notion of "frozen" variables that inhibits further instantiations. It achieves a predictable and easy-to-implement system. F_c can be viewed as freezing all arguments by default. QuickLook (QL) [Serrano et al. 2020] gathers information for all arguments, on top of the ordinary type inference procedure, for aiding in the type inference of the function head. If an instantiation is impredicative, it can only be applied if the target variable is guarded by an invariant type constructor to avoid the loss of principal types. Therefore ambiguous examples like single id are rejected by a guardedness check in QuickLook. Like all other local type inference techniques, F_c does not instantiate quantifiers of arguments (unless the argument itself is another application). Similar to QL, F_c does not instantiate types inside type constructors (including domain types of $\to$). A shared commonality between HMF, FreezeML, QuickLook and F_c is that they all faithfully adopt the type syntax of System F. In general, F_c and LTI are less ambitious in terms of type inference and aim instead at lighter type inference approaches, that can scale well to other more complex advanced type system features. Moreover, many works on ML-style polymorphism do not support explicit instantiations. F_c, like other works following local type inference, supports both implicit impredicative instantiations and explicit impredicative instantiations naturally.

Most work on ML-style first-class polymorphism do not deal with other forms of subtyping. SuperF [Parreaux et al. 2024] is an exception, which also considers intersection and union types. SuperF takes the inspiration from MLSub and algebraic subtyping [Dolan and Mycroft 2017; Parreaux 2020]. By making the subtyping relation form a distributive lattice, MLSub can reduce the subtyping constraint solving problem into bi-unification. However, the algebraic structure also means less flexibility in choosing features. For example, algebraic subtyping approaches must include very specific type features such as $\top$ and $\bot$ types, intersection and union types, and recursive types, as well as distributivity axioms in subtyping. Thus these approaches cannot be easily applied to type systems without those features, such as System F, and instead require modifications or extensions.

Predicative Higher-rank Polymorphism. Another important line of work is predicative Higher-Rank Polymorphism (HRP) [Dunfield and Krishnaswami 2013; Jones et al. 2007; Odersky and Läufer 1996]. They take another approach to achieve decidable subtyping by restricting the instantiation to monotypes, i.e., types that do not contain *any* universal types:

$$\frac{\Gamma \vdash_{\mathsf{mono}} \tau \quad \Gamma \vdash [\tau/\alpha]A <: B}{\Gamma \vdash \forall \alpha.\, A <: B}$$

They also allow inferring monotypes for fully unannotated functions. Their algorithms generally adopt unification-based techniques to solve these monotype instantiations. Although monotypes provide an elegant characterization of what types are guaranteed inferable, such a restriction makes these systems less expressive than System F. F^e [Zhao and Oliveira 2022] mitigates the loss of expressive power by allowing polytype instantiations to be explicitly annotated. It also adds $\top$ and $\bot$ types. Several works further extend F^e with other subtyping features. $F^b_{\leq}$ [Cui et al. 2023] adds bounded quantification and $F^e_{\sqcup\sqcap}$ [Jiang et al. 2025] adds intersection and union types to predicative HRP. However, while the type syntax is extended, the syntax of monotypes (i.e. the implicitly instantiable types) is still restricted to simple forms, limiting their power in inferring implicit instantiations. F_c, on the other hand, can infer impredicative instantiations by utilizing information from the surrounding context.

Matching in Type Inference. Matching [Huet 1976], also called one-sided unification [Bürckert 1986], is the problem of solving (in)equations where only one side has unknown variables to solve, and the other does not. This distinction was also identified by local type inference [Pierce and Turner 2000], but was rarely exploited by other type inference techniques afterwards. The major difference between unification and matching happens during solving equalities $\hat{\alpha} = A$. $\hat{\alpha}$ could occur free in A in the setting of unification, but never in the setting of matching. Usually, unification algorithms adopt an occurs-check requiring $\hat{\alpha} \notin \mathsf{FV}(A)$ to prevent non-termination. In simple type systems, the occurs-check does no harm since there cannot be a solution B for $\hat{\alpha}$ such that $B = [B/\hat{\alpha}]A$ if $\hat{\alpha} \in \mathsf{FV}(A)$. For example, $\hat{\alpha} = \hat{\alpha} \to \mathsf{Int}$ has no solutions with STLC types. Rejecting such patterns directly maintains a complete algorithm. However, with more complex subtyping, the occurs-check can wrongly reject (in)equations with solutions. For instance, $\hat{\alpha} \leq \hat{\alpha} \to \mathsf{Int}$ has solutions like $\hat{\alpha} = \bot$ (i.e. $\bot \leq \bot \to \mathsf{Int}$) in systems with $\top$ and $\bot$, or solutions like $\hat{\alpha} = \forall \beta.\, \beta$ in systems with universal types. The same issue arises with several other features, including intersection and union types [Barbanera et al. 1995; Barendregt et al. 1983; Compagnoni and Pierce 1996; Coppo and Dezani-Ciancaglini 1978; MacQueen et al. 1986; Pierce 1991; Pottinger 1980; Reynolds 1991], which are examples of features that lead to a non-structural subtype relation. Non-structural subtyping allows types with different constructors to be compared, making the occurs-check invalid. Without the occurs-check, there are no known complete algorithms for finding instantiations, and the decidability of subtyping remains unknown for most forms of non-structural subtyping [Dudenhefner et al. 2016; Su et al. 2002]. Unlike unification, matching is known to be decidable in many more settings, including the cases where the solution domain is extended with intersection types [Düdder et al. 2013], or the matching itself is higher-order [Stirling 2009].

8 Conclusion

Local contextual type inference offers a fresh path through the long-standing complexity of partial type inference. With contextual typing and CTAS, we sidestep the complex specifications of earlier LTI approaches. Our mechanization closes the gap between theory and practice, while the simplicity of F_c opens doors to scalable extensions like subtyping. We hope that, as modern languages grow increasingly ambitious in their type systems, local contextual type inference stands as a principled and practical foundation for inference, ready to meet that challenge.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments. We thank Marco Servetto for his helpful feedback on the manuscript. This work has been sponsored by Hong Kong Research Grant Council project number 17213525.

References

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and Union Types: Syntax and Semantics. *Information and Computation* 119, 2 (June 1995), 202–230.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983), 931–940. doi:10.2307/2273659

Roger Bosman, Georgios Karachalias, and Tom Schrijvers. 2023. No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:18. doi:10.4230/LIPICS.ITP.2023.8

Didier Le Botlan and Didier Rémy. 2003. ML^F: raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 27–38. doi:10.1145/944705.944709

Hans-Jürgen Bürckert. 1989. Matching - A Special Case of Unification? *J. Symb. Comput.* 8, 5 (1989), 523–536. doi:10.1016/S0747-7171(89)80057-4

Hans-Jürgen Bürckert. 1986. Some relationships between unification, restricted unification, and matching. In *8th International Conference on Automated Deduction*, Jörg H. Siekmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 514–524.

Jacek Chrząszcz. 1998. Polymorphic subtyping without distributivity. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–355.

Adriana B Compagnoni and Benjamin C Pierce. 1996. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science* 6, 5 (1996), 469–501.

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. 19 (01 1978), 139–156.

Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2083–2111. doi:10.1145/3622871

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 60–72. doi:10.1145/3009837.3009882

Boris Düdder, Moritz Martens, and Jakob Rehof. 2013. Intersection Type Matching with Subtyping. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7941)*, Masahito Hasegawa (Ed.). Springer, 125–139. doi:10.1007/978-3-642-38946-7_11

Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. 2016. The Intersection Type Unification Problem. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 52)*. 19:1–19:16.

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. doi:10.1145/3450952

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. doi:10.1145/2500365.2500582

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 423–437. doi:10.1145/3385412.3386003

Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169. doi:10.1006/INCO.1999.2830

Jean-Yves Girard. 1972. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. (1972).

Jacques Herbrand. 1930. *Recherches sur la théorie de la démonstration.* http://eudml.org/doc/192791

Haruo Hosoya and Benjamin Pierce. 1999. How Good is Local Type Inference? (07 1999).

Gérard P. Huet. 1976. Resolution d'Equations dans des Langages d'Order 1, 2. , ω. *PhD thesis, Universite de Paris VII* (1976). https://cir.nii.ac.jp/crid/1571980074398472192

Christopher Jenkins and Aaron Stump. 2018. Spine-local Type Inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018*, Matteo Cimini and Jay

McCarthy (Eds.). ACM, 37–48. doi:10.1145/3310232.3310233

Shengyi Jiang, Chen Cui, and Bruno C. d. S. Oliveira. 2025. Bidirectional Higher-Rank Polymorphism with Intersection and Union Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 2118–2148. doi:10.1145/3704907

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Funct. Program.* 17, 1 (2007), 1–82. doi:10.1017/S0956796806006034

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 283–294. doi:10.1145/1411204.1411245

Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 66–77. doi:10.1145/1480881.1480891

Paul Blain Levy. 2001. *Call-by-push-value.* Ph. D. Dissertation. Queen Mary University of London, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233

David MacQueen, Gordon Plotkin, and Ravi Sethi. 1986. An Ideal Model for Recursive Polymorphic Types. *Information and Control* 71 (1986), 92–130.

Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. 2022. Implicit Polarized F: local type inference for impredicativity. *CoRR* abs/2203.01835. arXiv:2203.01835 doi:10.48550/ARXIV.2203.01835

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. doi:10.1016/0022-0000(78)90014-4

John C. Mitchell. 1988. Polymorphic Type Inference and Containment. *Inf. Comput.* 76, 2/3 (1988), 211–249. doi:10.1016/0890-5401(88)90009-0

Martin Odersky and Konstantin Läufer. 1996. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, 54–67.

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 41–53. doi:10.1145/360204.360207

Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 124:1–124:28. doi:10.1145/3409006

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL (2024), 1418–1450. doi:10.1145/3632890

Benjamin C. Pierce. 1991. *Programming with intersection types, union types, and polymorphism.* Technical Report CMU-CS-91-106. Carnegie Mellon University.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. doi:10.1145/345099.345100

Hubert Plociniczak. 2016. *Decrypting Local Type Inference.* Ph. D. Dissertation. EPFL, Switzerland. doi:10.5075/EPFL-THESIS-6741

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577.

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard J. Robinet (Ed.). Springer, 408–423. doi:10.1007/3-540-06859-7_148

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 526)*, Takayasu Ito and Albert R. Meyer (Eds.). Springer, 675–700. doi:10.1007/3-540-54415-1_70

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41. doi:10.1145/321250.321253

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP (2020), 89:1–89:29. doi:10.1145/3408971

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 783–796. doi:10.1145/3192366.3192389

Colin Stirling. 2009. Decidability of higher-order matching. *Log. Methods Comput. Sci.* 5, 3 (2009). http://arxiv.org/abs/0907.3804

Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. 2002. The First-Order Theory of Subtyping Constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 203–216. doi:10.1145/503272.503292

Jerzy Tiuryn and Pawel Urzyczyn. 1996. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia Lawall (Eds.). ACM, 251–262. doi:10.1145/1159803.1159838

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 295–306. doi:10.1145/1411204.1411246

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 272–299. doi:10.1007/978-3-319-89884-1_10

Xu Xue, Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2025. *Local Contextual Type Inference (Artifact)*. doi:10.5281/zenodo.17491013

Xu Xue and Bruno C. d. S. Oliveira. 2024. Contextual Typing. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 880–908.

Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. doi:10.4230/LIPICS.ECOOP.2022.2

Received 2025-07-10; accepted 2025-11-06