Let Arguments Go First

Ningning Xie(✉) and Bruno C. d. S. Oliveira

The University of Hong Kong, Pokfulam, Hong Kong
{nnxie,bruno}@cs.hku.hk

Abstract. Bi-directional type checking has proved to be an extremely useful and versatile tool for type checking and type inference. The conventional presentation of bi-directional type checking consists of two modes: *inference* mode and *checked* mode. In traditional bi-directional type-checking, type annotations are used to guide (via the checked mode) the type inference/checking procedure to determine the type of an expression, and *type information flows from functions to arguments*.

This paper presents a variant of bi-directional type checking where the *type information flows from arguments to functions*. This variant retains the inference mode, but adds a so-called *application* mode. Such design can remove annotations that basic bi-directional type checking cannot, and is useful when type information from arguments is required to type-check the functions being applied. We present two applications and develop the meta-theory (mostly verified in Coq) of the application mode.

1 Introduction

Bi-directional type checking has been known in the folklore of type systems for a long time. It was popularized by Pierce and Turner's work on *local type inference* [29]. Local type inference was introduced as an alternative to Hindley-Milner (henceforth HM system) type systems [11,17], which could easily deal with polymorphic languages with subtyping. Bi-directional type checking is one component of local type inference that, aided by some type annotations, enables type inference in an expressive language with polymorphism and subtyping. Since Pierce and Turner's work, various other authors have proved the effectiveness of bi-directional type checking in several other settings, including many different systems with subtyping [12,14,15], systems with dependent types [2,3,10,21,37], and various other works [1,7,13,22,28]. Furthermore, bi-directional type checking has also been combined with HM-style techniques for providing type inference in the presence of higher-ranked types [14,27].

The key idea in bi-directional type checking is simple. In its basic form typing is split into *inference* and *checked* modes. The most salient feature of a bi-directional type-checker is when information deduced from inference mode is used to guide checking of an expression in checked mode. One of such interactions between modes happens in the typing rule for function applications:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \text{APP}$$

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 272–299, 2018.
https://doi.org/10.1007/978-3-319-89884-1_10

In the above rule, which is a standard bi-directional rule for checking applications, the two modes are used. First we synthesize ($\Rightarrow$) the type $A \to B$ from e_1, and then check ($\Leftarrow$) e_2 against A, returning B as the type for the application.

This paper presents a variant of bi-directional type checking that employs a so-called *application* mode. With the application mode the design of the application rule (for a simply typed calculus) is as follows:

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma \shortmid \Psi, A \vdash e_1 \;\Rightarrow\; A \to B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \;\Rightarrow\; B}\ \text{APP}$$

In this rule, there are two kinds of judgments. The first judgment is just the usual inference mode, which is used to infer the type of the argument e_2. The second judgment, the application mode, is similar to the inference mode, but it has an additional context Ψ. The context Ψ is a stack that tracks the types of the arguments of outer applications. In the rule for application, the type of the argument e_2 is inferred first, and then pushed into Ψ for inferring the type of e_1. Applications are themselves in the application mode, since they can be in the context of an outer application. With the application mode it is possible to infer the type for expressions such as `(λx. x) 1` without additional annotations.

Bi-directional type checking with an application mode may still require type annotations and it gives different trade-offs with respect to the checked mode in terms of type annotations. However the different trade-offs open paths to different designs of type checking/inference algorithms. To illustrate the utility of the application mode, we present two different calculi as applications. The first calculus is a higher ranked implicit polymorphic type system, which infers higher-ranked types, generalizes the HM type system, and has polymorphic **let** as syntactic sugar. As far as we are aware, no previous work enables an HM-style let construct to be expressed as syntactic sugar. For this calculus many results are proved using the Coq proof assistant [9], including type-safety. Moreover a sound and complete algorithmic system, inspired by Peyton Jones et al. [27], is also developed. A second calculus with *explicit polymorphism* illustrates how the application mode is compatible with type applications, and how it adds expressiveness by enabling an encoding of type declarations in a System-F-like calculus. For this calculus, all proofs (including type soundness), are mechanized in Coq.

We believe that, similarly to standard bi-directional type checking, bi-directional type checking with an application mode can be applied to a wide range of type systems. Our work shows two particular and non-trivial applications. Other potential areas of applications are other type systems with subtyping, static overloading, implicit parameters or dependent types.

In summary the contributions of this paper are[1]:

- **A variant of bi-directional type checking** where the inference mode is combined with a new, so-called, application mode. The application mode naturally propagates type information from arguments to the functions.
- **A new design for type inference of higher-ranked types** which generalizes the HM type system, supports a polymorphic **let** as syntactic sugar, and infers higher rank types. We present a syntax-directed specification, an elaboration semantics to System F, some meta-theory in Coq, and an algorithmic type system with completeness and soundness proofs.
- **A System-F-like calculus** as a theoretical response to the challenge noted by Pierce and Turner [29]. It shows that the application mode is compatible with type applications, which also enables encoding type declarations. We present a type system and meta-theory, including proofs of type safety and uniqueness of typing in Coq.

2 Overview

2.1 Background: Bi-directional Type Checking

Traditional type checking rules can be heavyweight on annotations, in the sense that lambda-bound variables always need explicit annotations. Bi-directional type checking [29] provides an alternative, which allows types to propagate downward the syntax tree. For example, in the expression $(\lambda \texttt{f:Int} \to \texttt{Int. f})\ (\lambda \texttt{y. y})$, the type of y is provided by the type annotation on `f`. This is supported by the bi-directional typing rule for applications:

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

Specifically, if we know that the type of e_1 is a function from $\texttt{A} \to \texttt{B}$, we can check that e_2 has type `A`. Notice that here the type information flows from functions to arguments.

One guideline for designing bi-directional type checking rules [15] is to distinguish introduction rules from elimination rules. Constructs which correspond to introduction forms are *checked* against a given type, while constructs corresponding to elimination forms *infer* (or synthesize) their types. For instance, under this design principle, the introduction rule for pairs is supposed to be in checked mode, as in the rule Pair-C.

$$\frac{\Gamma \vdash e_1 \Leftarrow A \qquad \Gamma \vdash e_2 \Leftarrow B}{\Gamma \vdash (e_1, e_2) \Leftarrow (A, B)}\ \textsc{Pair-C} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash (e_1, e_2) \Rightarrow (A, B)}\ \textsc{Pair-I}$$

[1] All supplementary materials are available in https://bitbucket.org/ningningxie/let-arguments-go-first.

Unfortunately, this means that the trivial program `(1, 2)` cannot type-check, which in this case has to be rewritten to `(1, 2) : (Int , Int)`.

In this particular case, bi-directional type checking goes against its original intention of removing burden from programmers, since a seemingly unnecessary annotation is needed. Therefore, in practice, bi-directional type systems do not strictly follow the guideline, and usually have additional inference rules for the introduction form of constructs. For pairs, the corresponding rule is Pair-I.

Now we can type check `(1, 2)`, but the price to pay is that two typing rules for pairs are needed. Worse still, the same criticism applies to other constructs. This shows one drawback of bi-directional type checking: often to minimize annotations, many rules are duplicated for having both inference and checked mode, which scales up with the typing rules in a type system.

2.2 Bi-directional Type Checking with the Application Mode

We propose a variant of bi-directional type checking with a new *application mode*. The application mode preserves the advantage of bi-directional type checking, namely many redundant annotations are removed, while certain programs can type check with even fewer annotations. Also, with our proposal, the inference mode is a special case of the application mode, so it does not produce duplications of rules in the type system. Additionally, the checked mode can still be *easily* combined into the system (see Sect. 5.1 for details). The essential idea of the application mode is to enable the type information flow in applications to propagate from arguments to functions (instead of from functions to arguments as in traditional bi-directional type checking).

To motivate the design of bi-directional type checking with an application mode, consider the simple expression

```
(λx. x) 1
```

This expression cannot type check in traditional bi-directional type checking because unannotated abstractions only have a checked mode, so annotations are required. For example, `((λx. x) : Int → Int) 1`.

In this example we can observe that if the type of the argument is accounted for in inferring the type of `λx. x`, then it is actually possible to deduce that the lambda expression has type `Int → Int` , from the argument `1`.

The Application Mode. If types flow from the arguments to the function, an alternative idea is to push the type of the arguments into the typing of the function, as the rule that is briefly introduced in Sect. 1:

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma \shortmid \Psi, A \vdash e_1 \;\Rightarrow\; A \to B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \;\Rightarrow\; B}\ \text{APP}$$

Here the argument e_2 synthesizes its type `A`, which then is pushed into the application context Ψ. Lambda expressions can now make use of the application context, leading to the following rule:

$$\frac{\Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; A \to B}\ \textsc{Lam}$$

The type `A` that appears last in the application context serves as the type for `x`, and type checking continues with a smaller application context and `x:A` in the typing context. Therefore, using the rule App and Lam, the expression `(λx. x) 1` can type-check without annotations, since the type `Int` of the argument `1` is used as the type of the binding `x`.

Note that, since the examples so far are based on simple types, obviously they can be solved by integrating type inference and relying on techniques like unification or constraint solving. However, here the point is that the application mode helps to reduce the number of annotations *without requiring such sophisticated techniques*. Also, the application mode helps with situations where those techniques cannot be easily applied, such as type systems with subtyping.

Interpretation of the Application Mode. As we have seen, the guideline for designing bi-directional type checking [15], based on introduction and elimination rules, is often not enough in practice. This leads to extra introduction rules in the inference mode. The application mode does not distinguish between introduction rules and elimination rules. Instead, to decide whether a rule should be in inference or application mode, we need to think whether the expression can be applied or not. Variables, lambda expressions and applications are all examples of expressions that can be applied, and they should have application mode rules. However pairs or literals cannot be applied and should have inference rules. For example, type checking pairs would simply lead to the rule Pair-I. Nevertheless elimination rules of pairs could have non-empty application contexts (see Sect. 5.2 for details). In the application mode, arguments are always inferred first in applications and propagated through application contexts. An empty application context means that an expression is not being applied to anything, which allows us to model the inference mode as a particular case[2].

Partial Type Checking. The inference mode synthesizes the type of an expression, and the checked mode checks an expression against some type. A natural question is how do these modes compare to application mode. An answer is that, in some sense: the application mode is stronger than inference mode, but weaker than checked mode. Specifically, the inference mode means that we know nothing about the type an expression before hand. The checked mode means that the whole type of the expression is already known before hand. With the application mode we know some partial type information about the type of an expression:

[2] Although the application mode generalizes the inference mode, we refer to them as two different modes. Thus the variant of bi-directional type checking in this paper is interpreted as a type system with both *inference* and *application* modes.

we know some of its argument types (since it must be a function type when the application context is non-empty), but not the return type.

Instead of nothing or all, this partialness gives us a finer grain notion on how much we know about the type of an expression. For example, assume $e : A \to B \to C$. In the inference mode, we only have e. In the checked mode, we have both e and $A \to B \to C$. In the application mode, we have e, and maybe an empty context (which degenerates into inference mode), or an application context A (we know the type of first argument), or an application context B, A (we know the types of both arguments).

Trade-offs. Note that the application mode is *not* conservative over traditional bidirectional type checking due to the different information flow. However, it provides a new design choice for type inference/checking algorithms, especially for those where the information about arguments is useful. Therefore we next discuss some benefits of the application mode for two interesting cases where functions are either variables; or lambda (or type) abstractions.

2.3 Benefits of Information Flowing from Arguments to Functions

Local Constraint Solver for Function Variables. Many type systems, including type systems with *implicit polymorphism* and/or *static overloading*, need information about the types of the arguments when type checking function variables. For example, in conventional functional languages with implicit polymorphism, function calls such as `(id 3)` where `id: ∀a. (a → a)`, are *pervasive*. In such a function call the type system must instantiate a to Int. Dealing with such implicit instantiation gets trickier in systems with *higher-ranked types*. For example, Peyton Jones et al. [27] require additional syntactic forms and relations, whereas Dunfield and Krishnaswami [14] add a special purpose *application judgment*.

With the application mode, all the type information about the arguments being applied is available in application contexts and can be used to solve instantiation constraints. To exploit such information, the type system employs a special subtyping judgment called *application subtyping*, with the form $\Psi \vdash A \leq B$. Unlike conventional subtyping, computationally Ψ and A are interpreted as inputs and B as output. In above example, we have that $\mathsf{Int} \vdash \forall a.a \to a \leq B$ and we can determine that $a = \mathsf{Int}$ and $B = \mathsf{Int} \to \mathsf{Int}$. In this way, type system is able to solve the constraints *locally* according to the application contexts since we no longer need to propagate the instantiation constraints to the typing process.

Declaration Desugaring for Lambda Abstractions. An interesting consequence of the usage of an application mode is that it enables the following **let** sugar:

$$\textbf{let}\ \texttt{x} = e_1\ \textbf{in}\ e_2 \rightsquigarrow (\lambda \texttt{x}.\ e_2)\ e_1$$

Such syntactic sugar for **let** is, of course, standard. However, in the context of implementations of typed languages it normally requires extra type annotations or a more sophisticated type-directed translation. Type checking $(\lambda x.\ e_2)\ e_1$

would normally require annotations (for example an annotation for x), or otherwise such annotation should be inferred first. Nevertheless, with the application mode no extra annotations/inference is required, since from the type of the argument e_1 it is possible to deduce the type of x. Generally speaking, with the application mode *annotations are never needed for applied lambdas*. Thus **let** can be the usual sugar from the untyped lambda calculus, including HM-style **let** expression and even type declarations.

2.4 Application 1: Type Inference of Higher-Ranked Types

As a first illustration of the utility of the application mode, we present a calculus with *implicit predicative higher-ranked polymorphism*.

Higher-Ranked Types. Type systems with higher-ranked types generalize the traditional HM type system, and are useful in practice in languages like Haskell or other ML-like languages. Essentially higher-ranked types enable much of the expressive power of System F, with the advantage of implicit polymorphism. Complete type inference for System F is known to be undecidable [36]. Therefore, several partial type inference algorithms, exploiting additional type annotations, have been proposed in the past instead [15,25,27,31].

Higher-Ranked Types and Bi-directional Type Checking. Bi-directional type checking is also used to help with the inference of higher-ranked types [14,27]. Consider the following program:

```
(λf. (f 1, f 'c')) (λx. x)
```

which is not typeable under those type systems because they fail to infer the type of `f`, since it is supposed to be polymorphic. Using bi-directional type checking, we can rewrite this program as

```
((λf. (f 1, f 'c')) : (∀a. a → a) → (Int, Char)) (λx . x)
```

Here the type of `f` can be easily derived from the type signature using checked mode in bi-directional type checking. However, although some redundant annotations are removed by bi-directional type checking, the burden of inferring higher-ranked types is still carried by programmers: they are forced to add polymorphic annotations to help with the type derivation of higher-ranked types. For the above example, the type annotation is still *provided by programmers*, even though the necessary type information can be derived intuitively without any annotations: `f` is applied to `λx. x`, which is of type `∀a. a → a`.

Generalization. Generalization is famous for its application in let polymorphism in the HM system, where generalization is adopted at let bindings. Let polymorphism is a useful component to introduce top-level quantifiers (rank 1 types) into a polymorphic type system. The previous example becomes typeable in the HM system if we rewrite it to: **let** `f = λx. x` **in** `(f 1, f 'c')`.

Type Inference for Higher-Ranked Types with the Application Mode. Using our bi-directional type system with an application mode, the original expression can type check without annotations or rewrites: `(λf. (f 1, f 'c')) (λx. x)`.

This result comes naturally if we allow type information flow from arguments to functions. For inferring polymorphic types for arguments, we use *generalization*. In the above example, we first infer the type `∀a. a → a` for the argument, then pass the type to the function. A nice consequence of such an approach is that HM-style polymorphic **let** expressions are simply regarded as syntactic sugar to a combination of lambda/application:

let `x =` e_1 **in** e_2 $\rightsquigarrow$ `(λx.` e_2`)` e_1

With this approach, nested lets can lead to types which are *more general* than HM. For example, **let** `s = λx. x` **in let** `t = λy. s` **in** `e`. The type of `s` is `∀a. a → a` after generalization. Because `t` returns `s` as a result, we might expect `t: ∀b. b → (∀a. a → a)`, which is what our system will return. However, HM will return type `t: ∀b. ∀a. b → (a → a)`, as it can only return rank 1 types, which is less general than the previous one according to Odersky and Läufer's subtyping relation for polymorphic types [24].

Conservativity over the Hindley-Milner Type System. Our type system is a conservative extension over the Hindley-Milner type system, in the sense that every program that can type-check in HM is accepted in our type system, which is explained in detail in Sect. 3.2. This result is not surprising: after desugaring **let** into a lambda and an application, programs remain typeable.

Comparing Predicative Higher-Ranked Type Inference Systems. We will give a full discussion and comparison of related work in Sect. 6. Among those works, we believe the work by Dunfield and Krishnaswami [14], and the work by Peyton Jones et al. [27] are the most closely related work to our system. Both their systems and ours are based on a *predicative* type system: universal quantifiers can only be instantiated by monotypes. So we would like to emphasize our system's properties in relation to those works. In particular, here we discuss two interesting differences, and also briefly (and informally) discuss how the works compare in terms of expressiveness.

(1) Inference of higher-ranked types. In both works, every polymorphic type inferred by the system must correspond to one annotation provided by the programmer. However, in our system, some higher-ranked types can be inferred from the expression itself without any annotation. The motivating expression above provides an example of this.
(2) Where are annotations needed? Since type annotations are useful for inferring higher rank types, a clear answer to the question where annotations are needed is necessary so that programmers know when they are required to write annotations. To this question, previous systems give a concrete answer: only on the binding of polymorphic types. Our answer is slightly different: only on the bindings of polymorphic types in abstractions *that are not applied to arguments*. Roughly speaking this means that our system ends up with fewer or smaller annotations.

(3) Expressiveness. Based on these two answers, it may seem that our system should accept all expressions that are typeable in their system. However, this is not true because the application mode is *not* conservative over traditional bi-directional type checking. Consider the expression `(λf : (∀a. a → a) → (Int, Char). f) (λg. (g 1, g 'a'))`, which is typeable in their system. In this case, even if g is a polymorphic binding without a type annotation the expression can still type-check. This is because the original application rule propagates the information from the outer binding into the inner expressions. Note that the fact that such expression type-checks does not contradict their guideline of providing type annotations for every polymorphic binder. Programmers that strictly follow their guideline can still add a polymorphic type annotation for g. However it does mean that it is a little harder to understand where annotations for polymorphic binders can be *omitted* in their system. This requires understanding how the applications in checked mode operate.

In our system the above expression is not typeable, as a consequence of the information flow in the application mode. However, following our guideline for annotations leads to a program that can be type-checked with a smaller annotation: `(λf. f) (λg : (∀a. a → a). (g 1, g 'a'))`. This means that our work is not conservative over their work, which is due to the design choice of the application typing rule. Nevertheless, we can always rewrite programs using our guideline, which often leads to fewer/smaller annotations.

2.5 Application 2: More Expressive Type Applications

The design choice of propagating arguments to functions was subject to consideration in the original work on local type inference [29], but was rejected due to possible non-determinism introduced by explicit type applications:

> *"It is possible, of course, to come up with examples where it would be beneficial to synthesize the argument types first and then use the resulting information to avoid type annotations in the function part of an application expression.... Unfortunately this refinement does not help infer the type of polymorphic functions. For example, we cannot uniquely determine the type of x in the expression* $(fun[X](x)\ e)$ [Int] 3." [29]

Therefore, as a response to this challenge, our second application is a variant of System F. Our development of the calculus shows that the application mode can actually work well with calculi with explicit type applications. To explain the new design, consider the expression:

```
(Λa. λx : a. x + 1) Int
```

which is not typeable in the traditional type system for System F. In System F the lambda abstractions do not account for the context of possible function applications. Therefore when type checking the inner body of the lambda abstraction, the expression `x + 1` is ill-typed, because all that is known is that `x` has the (abstract) type `a`.

If we are allowed to propagate type information from arguments to functions, then we can verify that `a = Int` and `x + 1` is well-typed. The key insight in the new type system is to use application contexts to track type equalities induced by type applications. This enables us to type check expressions such as the body of the lambda above (`x + 1`). Therefore, back to the problematic expression $(fun[X](x)\ e)$ [Int] 3, the type of `x` can be inferred as either `X` or `Int` since they are actually equivalent.

Sugar for Type Synonyms. In the same way that we can regard **let** expressions as syntactic sugar, in the new type system we further *gain built-in type synonyms for free*. A *type synonym* is a new name for an existing type. Type synonyms are common in languages such as Haskell. In our calculus a simple form of type synonyms can be desugared as follows:

type `a = A` **in** `e` $\rightsquigarrow$ `(Λa. e) A`

One practical benefit of such syntactic sugar is that it enables a direct encoding of a System F-like language with declarations (including type-synonyms). Although declarations are often viewed as a routine extension to a calculus, and are not formally studied, they are highly relevant in practice. Therefore, a more realistic formalization of a programming language should directly account for declarations. By providing a way to encode declarations, our new calculus enables a simple way to formalize declarations.

Type Abstraction. The type equalities introduced by type applications may seem like we are breaking System F type abstraction. However, we argue that *type abstraction* is still supported by our System F variant. For example:

let `inc = Λa. λx : a. x + 1` **in** `inc Int e`

(after desugaring) does *not* type-check, as in a System-F like language. In our type system lambda abstractions that are immediatelly applied to an argument, and unapplied lambda abstractions behave differently. Unapplied lambda abstractions are just like System F abstractions and retain type abstraction. The example above illustrates this. In contrast the typeable example `(Λa. λx : a. x + 1) Int`, which uses a lambda abstraction directly applied to an argument, can be regarded as the desugared expression for **type** `a = Int` **in** `λx : a . x + 1`.

3 A Polymorphic Language with Higher-Ranked Types

This section first presents a declarative, *syntax-directed* type system for a lambda calculus with implicit higher-ranked polymorphism. The interesting aspects about the new type system are: (1) the typing rules, which employ a combination of inference and application modes; (2) the novel subtyping relation under an application context. Later, we prove our type system is type-safe by a type directed translation to System F [16,27] in Sect. 3.4. Finally an algorithmic type system is discussed in Sect. 3.5.

3.1 Syntax

The syntax of the language is:

$$
\begin{array}{lr}
\text{Expr} & e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e_1\ e_2 \\
\text{Type} & A, B ::= a \mid A \to B \mid \forall a.A \mid \mathsf{Int} \\
\text{Monotype} & \tau ::= a \mid \tau_1 \to \tau_2 \mid \mathsf{Int} \\
\text{Typing Context} & \Gamma ::= \varnothing \mid \Gamma, x : A \\
\text{Application Context} & \Psi ::= \varnothing \mid \Psi, A
\end{array}
$$

Expressions. Expressions e include variables (x), integers (n), annotated lambda abstractions ($\lambda x : A.\ e$), lambda abstractions ($\lambda x.\ e$), and applications ($e_1\ e_2$). Letters x, y, z are used to denote term variables. Notably, the syntax does not include a **let** expression ($\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$). Let expressions can be regarded as the standard syntax sugar $(\lambda x.\ e_2)\ e_1$, as illustrated in more detail later.

Types. Types include type variables (a), functions ($A \to B$), polymorphic types ($\forall a.A$) and integers (Int). We use capital letters (A, B) for types, and small letters (a, b) for type variables. Monotypes are types without universal quantifiers.

Contexts. Typing contexts Γ are standard: they map a term variable x to its type A. We implicitly assume that all the variables in Γ are distinct. The main novelty lies in the *application contexts* Ψ, which are the main data structure needed to allow types to flow from arguments to functions. Application contexts are modeled as a stack. The stack collects the types of arguments in applications. The context is a stack because if a type is pushed last then it will be popped first. For example, inferring expression e under application context (a, Int), means e is now being applied to two arguments e_1, e_2, with $e_1 : \mathsf{Int}$, $e_2 : a$, so e should be of type $\mathsf{Int} \to a \to A$ for some A.

3.2 Type System

The top part of Fig. 1 gives the typing rules for our language. The judgment $\Gamma \shortmid \Psi \vdash e \Rightarrow B$ is read as: under typing context Γ, and application context Ψ, e has type B. The standard inference mode $\Gamma \vdash e \Rightarrow B$ can be regarded as a special case when the application context is empty. Note that the variable names are assumed to be fresh enough when new variables are added into the typing context, or when generating new type variables.

Rule T-Var says that if $x : A$ is in the typing context, and A is a subtype of B under application context Ψ, then x has type B. It depends on the subtyping rules that are explained in Sect. 3.3. Rule T-Int shows that integer literals are only inferred to have type Int under an empty application context. This is obvious since an integer cannot accept any arguments.

T-Lam shows the strength of application contexts. It states that, without annotations, if the application context is non-empty, a type can be popped from the application context to serve as the type for x. Inference of the body then continues with the rest of the application context. This is possible, because the

$$\boxed{\Gamma \mid \Psi \vdash e \;\Rightarrow\; B}$$

$$\frac{x : A \in \Gamma \qquad \Psi \vdash A \;<:\; B}{\Gamma \mid \Psi \vdash x \;\Rightarrow\; B}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash n \;\Rightarrow\; \mathsf{Int}}\ \text{T-Int}$$

$$\frac{\Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; A \to B}\ \text{T-Lam} \qquad \frac{\Gamma, x : \tau \vdash e \;\Rightarrow\; B}{\Gamma \vdash \lambda x.\ e \;\Rightarrow\; \tau \to B}\ \text{T-Lam2}$$

$$\frac{\Gamma, x : A \vdash e \;\Rightarrow\; B}{\Gamma \vdash \lambda x : A.\ e \;\Rightarrow\; A \to B}\ \text{T-LamAnn1}$$

$$\frac{C \;<:\; A \qquad \Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B}{\Gamma \mid \Psi, C \vdash \lambda x : A.\ e \;\Rightarrow\; C \to B}\ \text{T-LamAnn2} \qquad \frac{\bar{a} = ftv(A) - ftv(\Gamma)}{\Gamma_{gen}(A) = \forall \bar{a}.A}\ \text{T-Gen}$$

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \qquad \Gamma_{gen}(A) = B \qquad \Gamma \mid \Psi, B \vdash e_1 \;\Rightarrow\; B \to C}{\Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; C}\ \text{T-App}$$

$$\boxed{A \;<:\; B}$$

$$\frac{}{\mathsf{Int} \;<:\; \mathsf{Int}}\ \text{S-Int} \qquad \frac{}{a \;<:\; a}\ \text{S-Var} \qquad \frac{A \;<:\; B}{A \;<:\; \forall a.B}\ \text{S-ForallR}$$

$$\frac{A[\![a \mapsto \tau]\!] \;<:\; B}{\forall a.A \;<:\; B}\ \text{S-ForallL} \qquad \frac{C \;<:\; A \qquad B \;<:\; D}{A \to B \;<:\; C \to D}\ \text{S-Fun}$$

$$\boxed{\Psi \vdash A \;<:\; B}$$

$$\frac{}{\varnothing \vdash A \;<:\; A}\ \text{S-Empty} \qquad \frac{\Psi, C \vdash A[\![a \mapsto \tau]\!] \;<:\; B}{\Psi, C \vdash \forall a.A \;<:\; B}\ \text{S-ForallL2}$$

$$\frac{C \;<:\; A \qquad \Psi \vdash B \;<:\; D}{\Psi, C \vdash A \to B \;<:\; C \to D}\ \text{S-Fun2}$$

Fig. 1. Syntax-directed typing and subtyping.

expression $\lambda x.\ e$ is being applied to an argument of type A, which is the type at the top of the application context stack. Rule T-Lam2 deals with the case when the application context is empty. In this situation, a monotype τ is *guessed* for the argument, just like the Hindley-Milner system.

Rule T-LamAnn1 works as expected with an empty application context: a new variable x is put with its type A into the typing context, and inference continues on the abstraction body. If the application context is non-empty, then the rule T-LamAnn2 applies. It checks that C is a subtype of A before putting $x : A$ in the typing context. However, note that it is always possible to remove annotations in an abstraction if it has been applied to some arguments.

Rule T-App pushes types into the application context. The application rule first infers the type of the argument e_2 with type A. Then the type A is generalized in the same way that types in **let** expressions are generalized in the HM

type system. The resulting generalized type is B. The generalization is shown in rule T-Gen, where all free type variables are extracted to quantifiers. Thus the type of e_1 is now inferred under an application context extended with type B. The generalization step is important to infer higher ranked types: since B is a possibly polymorphic type, which is the argument type of e_1, then e_1 is of possibly a higher rank type.

***Let** Expressions.* The language does not have built-in **let** expressions, but instead supports **let** as syntactic sugar. The typing rule for **let** expressions in the HM system is (without the gray-shaded part):

$$\frac{\Gamma \vdash e_1 \;\Rightarrow\; A_1 \qquad \Gamma_{gen}(A_1) = A_2 \qquad \Gamma, x : A_2 \shaded{\mid \Psi} \vdash e_2 \;\Rightarrow\; B}{\Gamma \shaded{\mid \Psi} \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \;\Rightarrow\; B}\ \text{T-Let}$$

where we do generalization on the type of e_1, which is then assigned as the type of x while inferring e_2. Adapting this rule to our system with application contexts would result in the gray-shaded part, where the application context is only used for e_2, because e_2 is the expression being applied. If we desugar the **let** expression $(\textbf{let}\, x = e_1 \,\textbf{in}\, e_2)$ to $((\lambda x.\ e_2)\ e_1)$, we have the following derivation:

$$\frac{\Gamma \vdash e_1 \;\Rightarrow\; A_1 \qquad \Gamma_{gen}(A_1) = A_2 \qquad \dfrac{\Gamma, x : A_2 \mid \Psi \vdash e_2 \;\Rightarrow\; B}{\Gamma \mid \Psi, A_2 \vdash \lambda x.\ e_2 \;\Rightarrow\; A_2 \to B}\ \text{T-Lam}}{\Gamma \mid \Psi \vdash (\lambda x.\ e_2)\ e_1 \;\Rightarrow\; B}\ \text{T-App}$$

The type A_2 is now pushed into application context in rule T-App, and then assigned to x in T-Lam. Comparing this with the typing derivations with rule T-Let, we now have same preconditions. Thus we can see that the rules in Fig. 1 are sufficient to express an HM-style polymorphic let construct.

Meta-Theory. The type system enjoys several interesting properties, especially lemmas about application contexts. Before we present those lemmas, we need a helper definition of what it means to use arrows on application contexts.

Definition 1 ($\Psi \to B$). If $\Psi = A_1, A_2, ..., A_n$, then $\Psi \to B$ means the function type $A_n \to ... \to A_2 \to A_1 \to B$.

Such definition is useful to reason about the typing result with application contexts. One specific property is that the application context determines the form of the typing result.

Lemma 1 (Ψ Coincides with Typing Results). *If $\Gamma \mid \Psi \vdash e \;\Rightarrow\; A$, then for some A', we have $A = \Psi \to A'$.*

Having this lemma, we can always use the judgment $\Gamma \mid \Psi \vdash e \;\Rightarrow\; \Psi \to A'$ instead of $\Gamma \mid \Psi \vdash e \;\Rightarrow\; A$.

In traditional bi-directional type checking, we often have one subsumption rule that transfers between inference and checked mode, which states that if an

expression can be inferred to some type, then it can be checked with this type. In our system, we regard the normal inference mode $\Gamma \vdash e \Rightarrow A$ as a special case, when the application context is empty. We can also turn from normal inference mode into application mode with an application context.

Lemma 2 (Subsumption). *If* $\Gamma \vdash e \Rightarrow \Psi \to A$, *then* $\Gamma \shortmid \Psi \vdash e \Rightarrow \Psi \to A$.

The relationship between our system and standard Hindley Milner type system can be established through the desugaring of let expressions. Namely, if e is typeable in Hindley Milner system, then the desugared expression $|e|$ is typeable in our system, with a more general typing result.

Lemma 3 (Conservative over HM). *If* $\Gamma \vdash^{HM} e \Rightarrow A$, *then for some* B, *we have* $\Gamma \vdash |e| \Rightarrow B$, *and* $B <: A$.

3.3 Subtyping

We present our subtyping rules at the bottom of Fig. 1. Interestingly, our subtyping has two different forms.

Subtyping. The first judgment follows Odersky and Läufer [24]. $A <: B$ means that A is more polymorphic than B and, equivalently, A is a subtype of B. Rules S-Int and S-Var are trivial. Rule S-ForallR states A is subtype of $\forall a.B$ only if A is a subtype of B, with the assumption a is a fresh variable. Rule S-ForallL says $\forall a.A$ is a subtype of B if we can instantiate it with some τ and show the result is a subtype of B. In rule S-Fun, we see that subtyping is contra-variant on the argument type, and covariant on the return type.

Application Subtyping. The typing rule T-Var uses the second subtyping judgment $\Psi \vdash A <: B$. To motivate this new kind of judgment, consider the expression `id 1` for example, whose derivation is stuck at T-Var (here we assume $\mathsf{id} : \forall a.a \to a \in \Gamma$):

$$\dfrac{\Gamma \vdash 1 \Rightarrow \mathsf{Int} \qquad \Gamma_{gen}(\mathsf{Int}) = \mathsf{Int} \qquad \dfrac{\mathsf{id} : \forall a.a \to a \in \Gamma \qquad ???}{\Gamma \shortmid \mathsf{Int} \vdash \mathsf{id} \Rightarrow}\ \text{T-Var}}{\Gamma \vdash \mathsf{id}\ 1 \Rightarrow}\ \text{T-App}$$

Here we know that $\mathsf{id} : \forall a.a \to a$ and also, from the application context, that id is applied to an argument of type Int. Thus we need a mechanism for solving the instantiation $a = \mathsf{Int}$ and return a supertype $\mathsf{Int} \to \mathsf{Int}$ as the type of id. This is precisely what the application subtyping achieves: resolve instantiation constraints according to the application context. Notice that unlike existing works [14,27], application subtyping provides a way to solve instantiation more *locally*, since it does not mutually depend on typing.

Back to the rules in Fig. 1, one way to understand the judgment $\Psi \vdash A <: B$ from a computational point-of-view is that the type B is a *computed* output, rather than an input. In other words B is determined from Ψ and A. This is

unlike the judgment $A <: B$, where both A and B would be computationally interpreted as inputs. Therefore it is not possible to view $A <: B$ as a special case of $\Psi \vdash A <: B$ where Ψ is empty.

There are three rules dealing with application contexts. Rule S-Empty is for case when the application context is empty. Because it is empty, we have no constraints on the type, so we return it back unchanged. Note that this is where HM systems (also Peyton Jones et al. [27]) would normally use a rule Inst to remove top-level quantifiers:

$$\frac{}{\forall \bar{a}.A <: A[\![\bar{a} \mapsto \bar{\tau}]\!]}\ \textsc{Inst}$$

Our system does not need Inst, because in applications, type information flows from arguments to the function, instead of function to arguments. In the latter case, Inst is needed because a function type is wanted instead of a polymorphic type. In our approach, instantiation of type variables is avoided unless necessary.

The two remaining rules apply when the application context is non-empty, for polymorphic and function types respectively. Note that we only need to deal with these two cases because Int or type variables a cannot have a non-empty application context. In rule S-Forall2, we instantiate the polymorphic type with some τ, and continue. This instantiation is forced by the application context. In rule S-Fun2, one function of type $A \to B$ is now being applied to an argument of type C. So we check $C <: A$. Then we continue with B and the rest application context, and return $C \to D$ as the result type of the function.

Meta-Theory. Application subtyping is novel in our system, and it enjoys some interesting properties. For example, similarly to typing, the application context decides the form of the supertype.

Lemma 4 (Ψ Coincides with Subtyping Results). *If $\Psi \vdash A <: B$, then for some B', $B = \Psi \to B'$.*

Therefore we can always use the judgment $\Psi \vdash A <: \Psi \to B'$, instead of $\Psi \vdash A <: B$. Application subtyping is also reflexive and transitive. Interestingly, in those lemmas, if we remove all applications contexts, they are exactly the reflexivity and transitivity of traditional subtyping.

Lemma 5 (Reflexivity). $\Psi \vdash \Psi \to A <: \Psi \to A$.

Lemma 6 (Transitivity). *If $\Psi_1 \vdash A <: \Psi_1 \to B$, and $\Psi_2 \vdash B <: \Psi_2 \to C$, then $\Psi_2, \Psi_1 \vdash A <: \Psi_1 \to \Psi_2 \to C$.*

Finally, we can convert between subtyping and application subtyping. We can remove the application context and still get a subtyping relation:

Lemma 7 ($\Psi \vdash <:$ to $<:$). *If $\Psi \vdash A <: B$, then $A <: B$.*

Transferring from subtyping to application subtyping will result in a more general type.

Lemma 8 ($<:$ to $\Psi \vdash\ <:$). *If $A\ <:\ \Psi \to B_1$, then for some B_2, we have $\Psi \vdash A\ <:\ \Psi \to B_2$, and $B_2\ <:\ B_1$.*

This lemma may not seem intuitive at first glance. Consider a concrete example $\mathsf{Int} \to \forall a.a\ <:\ \mathsf{Int} \to \mathsf{Int}$, and $\mathsf{Int} \vdash \mathsf{Int} \to \forall a.a\ <:\ \mathsf{Int} \to \forall a.a$. The former one, holds because we have $\forall a.a\ <:\ \mathsf{Int}$ in the return type. But in the latter one, after Int is consumed from application context, we eventually reach S-Empty, which always returns the original type back.

3.4 Translation to System F, Coherence and Type-Safety

We translate the source language into a variant of System F that is also used in Peyton Jones et al. [27]. The translation is shown to be coherent and type safe. Due to space limitations, we only summarize the key aspects of the translation. Full details can be found in the supplementary materials of the paper.

The syntax of our target language is as follows:

$$\text{Expressions } s, f ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid s_1\ A$$

In the translation, we use f to refer to the coercion function produced by the subtyping translation, and s to refer to the translated term in System F. We write $\Gamma \vdash^F s : A$ to mean the term s has type A in System F.

The type-directed translation follows the rules in Fig. 1, with a translation output in the forms of judgments. We summarize all judgments as:

Judgment	Translation Output	Soundness
$A\ <:\ B\ \rightsquigarrow f$	coercion function f	$\varnothing \vdash^F f : A \to B$
$\Psi \vdash A\ <:\ B\ \rightsquigarrow f$	coercion function f	$\varnothing \vdash^F f : A \to B$
$\Gamma \mid \Psi \vdash e\ \Rightarrow\ A\ \rightsquigarrow s$	target expression s	$\Gamma \vdash^F s : A$

For example, $A\ <:\ B\ \rightsquigarrow f$ means that if $A\ <:\ B$ holds in the source language, we can translate it into a System F term f, which is a coercion function and has type $A \to B$. We prove that our system is type safe by proving that the translation produces well-typed terms.

Lemma 9 (Typing Soundness). *If $\Gamma \mid \Psi \vdash e\ \Rightarrow\ A\ \rightsquigarrow s$, then $\Gamma \vdash^F s : A$.*

However, there could be multiple targets corresponding to one expression due to the multiple choices for τ. To prove that the translation is coherent, we prove that all the translations for one expression have the same operational semantics. We write $|e|$ for the expressions after type erasure since types are useless after type checking. Because multiple targets could have different number of coercion functions, we use η-id equality [5] instead of syntactic equality, where two expressions are regarded as equivalent if they can turn into the same expression through η-reduction or removal of redundant identity functions. We then prove that our translation actually generates a *unique* target:

Lemma 10 (Coherence). *If $\Gamma_1 \mid \Psi_1 \vdash e \Rightarrow A\ \rightsquigarrow s_1$, and $\Gamma_2 \mid \Psi_2 \vdash e \Rightarrow B\ \rightsquigarrow s_2$, then $|s_1| \rightsquigarrow_{\eta id} |s_2|$.*

3.5 Algorithmic System

Even though our specification is syntax-directed, it does not directly lead to an algorithm, because there are still many guesses in the system, such as in rule T-Lam2. This subsection presents a brief introduction of the algorithm, which essentially follows the approach by Peyton Jones et al. [27]. Full details can be found in the supplementary materials.

Instead of guessing, the algorithm creates meta type variables $\widehat{\alpha}, \widehat{\beta}$ which are waiting to be solved. The judgment for the algorithmic type system is $(S_0, N_0) \shortmid \Gamma \shortmid \Psi \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$. Here we use N as name supply, from which we can always extract new names. We use S as a notation for the substitution that maps meta type variables to their solutions. For example, rule T-Lam2 becomes

$$\frac{(S_0, N_0) \shortmid \Gamma, x : \widehat{\beta} \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)}{(S_0, N_0\widehat{\beta}) \shortmid \Gamma \vdash \lambda x.\ e \Rightarrow \widehat{\beta} \to A \hookrightarrow (S_1, N_1)}\ \text{AT-Lam1}$$

Comparing it to rule T-Lam2, τ is replaced by a new meta type variable $\widehat{\beta}$ from name supply $N_0\widehat{\beta}$. But despite of the name supply and substitution, the rule retains the structure of T-Lam2.

Having the name supply and substitutions, the algorithmic system is a direct extension of the specification in Fig. 1, with a process to do unifications that solve meta type variables. Such unification process is quite standard and similar to the one used in the Hindley-Milner system. We proved our algorithm is sound and complete with respect to the specification.

Theorem 1 (Soundness). *If* $([], N_0) \shortmid \Gamma \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$*, then for any substitution* V *with* $dom(V) = fmv\ (S_1\Gamma, S_1A)$*, we have* $VS_1\Gamma \vdash e \Rightarrow VS_1A$.

Theorem 2 (Completeness). *If* $\Gamma \vdash e \Rightarrow A$*, then for a fresh* N_0*, we have* $([], N_0) \shortmid \Gamma \vdash e \Rightarrow B \hookrightarrow (S_1, N_1)$*, and for some* S_2*, we have* $\overline{\Gamma(S_2S_1B)} <: \overline{\Gamma(A)}$.

4 More Expressive Type Applications

This section presents a System-F-like calculus, which shows that the application mode not only does work well for calculi with explicit type applications, but it also adds interesting expressive power, while at the same time retaining uniqueness of types for *explicitly* polymorphic functions. One additional novelty in this section is to present another possible variant of typing and subtyping rules for the application mode, by exploiting the lemmas presented in Sects. 3.2 and 3.3.

$$\langle \emptyset \rangle A = A \qquad \langle \Gamma, x : B \rangle A = \langle \Gamma \rangle A$$
$$\langle \Gamma, a \rangle A = \langle \Gamma \rangle A \qquad \langle \Gamma, a = B \rangle A = \langle \Gamma \rangle (A[\![a \mapsto B]\!])$$

Fig. 2. Apply contexts as substitutions on types.

$$\frac{a \in \Gamma}{\Gamma \vdash a}\ \text{WF-TVar} \qquad \frac{}{\Gamma \vdash \mathsf{Int}}\ \text{WF-Int} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \text{WF-Arrow} \qquad \frac{\Gamma, a \vdash A}{\Gamma \vdash \forall a.A}\ \text{WF-All}$$

Fig. 3. Well-formedness.

4.1 Syntax

We focus on a new variant of the standard System F. The syntax is as follows:

$$\begin{array}{ll} \text{Expr} & e ::= x \mid n \mid \lambda x : A.\ e \mid \lambda x.\ e \mid e_1\ e_2 \mid \Lambda a.e \mid e\ [A] \\ \text{Type} & A ::= a \mid \mathsf{Int} \mid A \to B \mid \forall a.A \\ \text{Typing Context} & \Gamma ::= \varnothing \mid \Gamma, x : A \mid \Gamma, a \mid \Gamma, a = A \\ \text{Application Context} & \Psi ::= \varnothing \mid \Psi, A \mid \Psi, [A] \end{array}$$

The syntax is mostly standard. Expressions include variables x, integers n, annotated abstractions $\lambda x : A.\ s$, unannotated abstractions $\lambda x.\ e$, applications $e_1\ e_2$, type abstractions $\Lambda a.s$, and type applications $e_1\ [A]$. Types includes type variable a, integers Int, function types $A \to B$, and polymorphic types $\forall a.A$.

The main novelties are in the typing and application contexts. Typing contexts contain the usual term variable typing $x : A$, type variables a, and type equations $a = A$, which track equalities and are not available in System F. Application contexts use A for the *argument type* for term-level applications, and use $[A]$ for the *type argument itself* for type applications.

Applying Contexts. The typing contexts contain type equations, which can be used as substitutions. For example, $a = \mathsf{Int}, x : \mathsf{Int}, b = \mathit{Bool}$ can be applied to $a \to b$ to get the function type $\mathsf{Int} \to \mathit{Bool}$. We write $\langle \Gamma \rangle A$ for Γ applied as a substitution to type A. The formal definition is given in Fig. 2.

Well-Formedness. The type well-formedness under typing contexts is given in Fig. 3, which is quite straightforward. Notice that there is no rule corresponding to type variables in type equations. For example, a is not a well-formed type under typing context $a = \mathsf{Int}$, instead, $\langle a = \mathsf{Int} \rangle a$ is. In other words, we keep the invariant: *types are always fully substituted under the typing context.*

The well-formedness of typing contexts $\Gamma\ \mathit{ctx}$, and the well-formedness of application contexts $\Gamma \vdash \Psi$ can be defined naturally based on the well-formedness of types. The specific definitions can be found in the supplementary materials.

$$\boxed{\Gamma \shortmid \Psi \vdash e \Rightarrow B}$$

$$\frac{\Gamma \ ctx \qquad \Gamma \vdash \Psi \qquad x : A \in \Gamma \qquad \Psi \vdash A <: B}{\Gamma \shortmid \Psi \vdash x \Rightarrow B}\ \text{SF-Var} \qquad \frac{\Gamma \ ctx}{\Gamma \vdash n \Rightarrow \mathsf{Int}}\ \text{SF-Int}$$

$$\frac{\Gamma, x : \langle \Gamma \rangle A \vdash e \Rightarrow B}{\Gamma \vdash \lambda x : A.\ e \Rightarrow \langle \Gamma \rangle A \to B}\ \text{SF-LamAnn1}$$

$$\frac{\Gamma, x : \langle \Gamma \rangle A \shortmid \Psi \vdash e \Rightarrow B}{\Gamma \shortmid \Psi, \langle \Gamma \rangle A \vdash \lambda x : A.\ e \Rightarrow B}\ \text{SF-LamAnn2} \qquad \frac{\Gamma, x : A \shortmid \Psi \vdash e \Rightarrow B}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \Rightarrow B}\ \text{SF-Lam}$$

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \shortmid \Psi, A \vdash e_1 \Rightarrow B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \Rightarrow B}\ \text{SF-App} \qquad \frac{\Gamma, a \vdash e \Rightarrow B}{\Gamma \vdash \Lambda a.e \Rightarrow \forall a.B}\ \text{SF-TLam1}$$

$$\frac{\Gamma, a = A \shortmid \Psi \vdash e \Rightarrow B}{\Gamma \shortmid \Psi, [A] \vdash \Lambda a.e \Rightarrow B}\ \text{SF-TLam2} \qquad \frac{\Gamma \shortmid \Psi, [\langle \Gamma \rangle A] \vdash e \Rightarrow B}{\Gamma \shortmid \Psi \vdash e\ [A] \Rightarrow B}\ \text{SF-TApp}$$

$$\boxed{\Psi \vdash A <: B}$$

$$\frac{}{\varnothing \vdash A <: A}\ \text{SF-SEmpty}$$

$$\frac{\Psi \vdash B[\![a \mapsto A]\!] <: C}{\Psi, [A] \vdash \forall a.B <: C}\ \text{SF-STApp} \qquad \frac{\Psi \vdash B <: C}{\Psi, A \vdash A \to B <: C}\ \text{SF-SApp}$$

Fig. 4. Type system for the new System F variant.

4.2 Type System

Typing Judgments. From Lemmas 1 and 4, we know that the application context always coincides with typing/subtyping results. This means that the types of the arguments can be recovered from the application context. So instead of the whole type, we can use only the return type as the output type. For example, we review the rule T-Lam in Fig. 1:

$$\frac{\Gamma, x : A \shortmid \Psi \vdash e \Rightarrow B}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \Rightarrow A \to B}\ \text{T-Lam} \qquad \frac{\Gamma, x : A \shortmid \Psi \vdash e \Rightarrow C}{\Gamma \shortmid \Psi, A \vdash \lambda x.\ e \Rightarrow C}\ \text{T-Lam-Alt}$$

We have $B = \Psi \to C$ for some C by Lemma 1. Instead of B, we can directly return C as the output type, since we can derive from the application context that e is of type $\Psi \to C$, and $\lambda x.\ e$ is of type $(\Psi, A) \to C$. Thus we obtain the T-Lam-Alt rule.

Note that the choice of the style of the rules is only a matter of taste in the language in Sect. 3. However, it turns out to be very useful for our variant of System F, since it helps avoiding introducing types like $\forall a = \mathsf{Int}.a$. Therefore, we adopt the new form of judgment. Now the judgment $\Gamma \shortmid \Psi \vdash e \Rightarrow A$ is interpreted as: *under the typing context* Γ, *and the application context* Ψ, *the return type of* e *applied to the arguments whose types are in* Ψ *is* A.

Typing Rules. Using the new interpretation of the typing judgment, we give the typing rules in the top of Fig. 4. SF-Var depends on the subtyping rules. Rule SF-Int always infers integer types. Rule SF-LamAnn1 first applies current context on A, then puts $x : \langle \Gamma \rangle A$ into the typing context to infer e. The return type is a function type because the application context is empty. Rule SF-LamAnn2 has a non-empty application context, so it requests that the type at the top of the application context is equivalent to $\langle \Gamma \rangle A$. The output type is B instead of a function type. Notice how the invariant that types are fully substituted under the typing context is preserved in these two rules.

Rule SF-Lam pops the type A from the application context, puts $x : A$ into the typing context, and returns only the return type B. In rule SF-App, the argument type A is pushed into the application context for inferring e_1, so the output type B is the type of e_1 under application context (Ψ, A), which is exactly the return type of $e_1\ e_2$ under Ψ.

Rule SF-TLam1 is for type abstractions. The type variable a is pushed into the typing context, and the return type is a polymorphic type. In rule SF-TLam2, the application context has the type argument A at its top, which means the type abstraction is applied to A. We then put the type equation $a = A$ into the typing context to infer e. Like term-level applications, here we only return the type B instead of a polymorphic type. In rule SF-TApp, we first apply the typing context on the type argument A, then we put the applied type argument $\langle \Gamma \rangle A$ into the application context to infer e, and return B as the output type.

Subtyping. The definition of subtyping is given at the bottom of Fig. 4. As with the typing rules, the part of argument types corresponding to the application context is omitted in the output. We interpret the rule form $\Psi \vdash A <: B$ as, under the application context Ψ, A is a subtype of the type whose type arguments are Ψ and the return type is B.

Rule SF-SEmpty returns the input type under the empty application context. Rule SF-STApp instantiates a with the type argument A, and returns C. Note how application subtyping can be extended naturally to deal with type applications. Rule SF-SApp requests that the argument type is the same as the top type in the application context, and returns C.

4.3 Meta Theory

Applying the idea of the application mode to System F results in a well-behaved type system. For example, subtyping transitivity becomes more concise:

Lemma 11 (Subtyping transitivity). *If* $\Psi_1 \vdash A <: B$, *and* $\Psi_2 \vdash B <: C$, *then* $\Psi_2, \Psi_1 \vdash A <: C$.

Also, we still have the interesting subsumption lemma that transfers from the inference mode to the application mode:

Lemma 12 (Subsumption). *If* $\Gamma \vdash e \Rightarrow A$, *and* $\Gamma \vdash \Psi$, *and* $\Psi \vdash A <: B$, *then* $\Gamma \shortmid \Psi \vdash e \Rightarrow B$.

Furthermore, we prove the type safety by proving the progress lemma and the preservation lemma. The detailed definitions of operational semantics and values can be found in the supplementary materials.

Lemma 13 (Progress). *If* $\varnothing \vdash e \Rightarrow T$*, then either* e *is a value, or there exists* e'*, such that* $e \longrightarrow e'$*.*

Lemma 14 (Preservation). *If* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$*, and* $e \longrightarrow e'$*, then* $\Gamma \shortmid \Psi \vdash e' \Rightarrow A$*.*

Moreover, introducing type equality preserves unique types:

Lemma 15 (Uniqueness of typing). *If* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$*, and* $\Gamma \shortmid \Psi \vdash e \Rightarrow B$*, then* $A = B$*.*

5 Discussion

This section discusses possible design choices regarding bi-directional type checking with the application mode, and talks about possible future work.

5.1 Combining Application and Checked Modes

Although the application mode provides us with alternative design choices in a bi-directional type system, a checked mode can still be *easily* added. One motivation for the checked mode would be annotated expressions $e : A$, where the type of expressions is known and is therefore used to check expressions.

Consider adding $e : A$ for introducing the third checked mode for the language in Sect. 3. Notice that, since the checked mode is stronger than application mode, when entering checked mode the application context is no longer useful. Instead we use application subtyping to satisfy the application context requirements. A possible typing rule for annotation expressions is:

$$\frac{\Psi \vdash A <: B \qquad \Gamma \vdash e \Leftarrow A}{\Gamma \shortmid \Psi \vdash (e : A) \Rightarrow B} \text{ T-Ann}$$

Here, e is checked using its annotation A, and then we instantiate A to B using subtyping with application context Ψ.

Now we can have a rule set of the checked mode for all expressions. For example, one useful rule for abstractions in checked mode could be Abs-Chk, where the parameter type A serves as the type of x, and typing checks the body with B. Also, combined with the information flow, the checked rule for application checks the function with the full type.

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\ e \Leftarrow A \to B} \text{ Abs-Chk} \qquad \frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \vdash e_1 \Leftarrow A \to B}{\Gamma \vdash e_1\ e_2 \Leftarrow B} \text{ App-Chk}$$

Note that adding expression annotations might bring convenience for programmers, since annotations can be more freely placed in a program. For example, `(λf. f 1) : (Int → Int) → Int` becomes valid. However this does not add expressive power, since programs that are typeable under expression annotations, would remain typeable after moving the annotations to bindings. For example the previous program is equivalent to `(λf : (Int → Int). f 1)`.

This discussion is a sketch. We have not defined the corresponding declarative system nor algorithm. However we believe that the addition of a checked mode will *not* bring surprises to the meta-theory.

5.2 Additional Constructs

In this section, we show that the application mode is compatible with other constructs, by discussing how to add support for pairs in the language given in Sect. 3. A similar methodology would apply to other constructs like sum types, data types, if-then-else expressions and so on.

The introduction rule for pairs must be in the inference mode with an empty application context. Also, the subtyping rule for pairs is as expected.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B}{\Gamma \vdash (e_1, e_2) \Rightarrow (A, B)}\ \text{T-Pair} \qquad \frac{A_1 <: B_1 \qquad A_2 <: B_2}{(A_1, A_2) <: (B_1, B_2)}\ \text{S-Pair}$$

The application mode can apply to the elimination constructs of pairs. If one component of the pair is a function, for example, (**fst** $(\lambda x.\ x, 3)$ 4), then it is possible to have a judgment with a non-empty application context. Therefore, we can use the application subtyping to account for the application contexts:

$$\frac{\Gamma \vdash e \Rightarrow (A, B) \qquad \Psi \vdash A <: C}{\Gamma \shortmid \Psi \vdash \mathbf{fst}\ e \Rightarrow C}\ \text{T-Fst1} \qquad \frac{\Gamma \vdash e \Rightarrow (A, B) \qquad \Psi \vdash B <: C}{\Gamma \shortmid \Psi \vdash \mathbf{snd}\ e \Rightarrow C}\ \text{T-Snd1}$$

However, in polymorphic type systems, we need to take the subsumption rule into consideration. For example, in the expression $(\lambda x : (\forall a.(a, b)).\ \mathbf{fst}\ x)$, **fst** is applied to a polymorphic type. Interestingly, instead of a non-deterministic subsumption rule, having polymorphic types actually leads to a simpler solution. According to the philosophy of the application mode, the types of the arguments always flow into the functions. Therefore, instead of regarding (**fst** e) as an expression form, where e is itself an argument, we could regard **fst** as a function on its own, whose type is $(\forall ab.(a, b) \to a)$. Then as in the variable case, we use the subtyping rule to deal with application contexts. Thus the typing rules for **fst** and **snd** can be modeled as:

$$\frac{\Psi \vdash (\forall ab.(a, b) \to a) <: A}{\Gamma \shortmid \Psi \vdash \mathbf{fst} \Rightarrow A}\ \text{T-Fst2} \qquad \frac{\Psi \vdash (\forall ab.(a, b) \to b) <: A}{\Gamma \shortmid \Psi \vdash \mathbf{snd} \Rightarrow A}\ \text{T-Snd2}$$

Note that another way to model those two rules would be to simply have an initial typing environment $\Gamma_{initial} \equiv \mathbf{fst} : (\forall ab.(a, b) \to a), \mathbf{snd} : (\forall ab.(a, b) \to b)$. In this case the elimination of pairs be dealt directly by the rule for variables.

An extended version of the calculus presented in Sect. 3, which includes the rules for pairs (T-Pair, S-Pair, T-Fst2 and T-Snd2), has been formally studied. All the theorems presented in Sect. 3 hold with the extension of pairs.

5.3 Dependent Type Systems

One remark about the application mode is that the same idea is possibly applicable to systems with advanced features, where type inference is sophisticated or even undecidable. One promising application is, for instance, dependent type systems [2,3,10,21,37]. Type systems with dependent types usually unify the syntax for terms and types, with a single lambda abstraction generalizing both type and lambda abstractions. Unfortunately, this means that the **let** desugar is not valid in those systems. As a concrete example, consider desugaring the expression $\textbf{let}\, a = \mathsf{Int}\, \textbf{in}\, \lambda x : a.\ x + 1$ into `(λa. λx : a. x + 1) Int`, which is ill-typed because the type of x in the abstraction body is a and not Int.

Because **let** cannot be encoded, declarations cannot be encoded either. Modeling declarations in dependently typed languages is a subtle matter, and normally requires some additional complexity [34].

We believe that the same technique presented in Sect. 4 can be adapted into a dependently typed language to enable a **let** encoding. In a dependent type system with unified syntax for terms and types, we can combine the two forms in the typing context ($x : A$ and $a = A$) into a unified form $x = e : A$. Then we can combine two application rules SF-App and SF-TApp into De-App, and also two abstraction rules SF-Lam and SF-TLam1 into De-Lam.

$$\frac{\Gamma \vdash e_2 \ \Rightarrow\ A \qquad \Gamma \shortmid \Psi, e_2 : A \vdash e_1 \ \Rightarrow\ B}{\Gamma \shortmid \Psi \vdash e_1\ e_2 \ \Rightarrow\ B}\ \textsc{De-App} \qquad \frac{\Gamma, x = e_1 : A \shortmid \Psi \vdash e \ \Rightarrow\ B}{\Gamma \shortmid \Psi, e_1 : A \vdash \lambda x.\ e \ \Rightarrow\ B}\ \textsc{De-Lam}$$

With such rules it would be possible to handle declarations easily in dependent type systems. Note this is still a rough idea and we have not fully worked out the typing rules for this type system yet.

6 Related Work

6.1 Bi-directional Type Checking

Bi-directional type checking was popularized by the work of Pierce and Turner [29]. It has since been applied to many type systems with advanced features. The alternative application mode introduced by us enables a variant of bi-directional type checking. There are many other efforts to refine bi-directional type checking.

Colored local type inference [25] refines local type inference for *explicit* polymorphism by propagating partial type information. Their work is built on distinguishing inherited types (known from the context) and synthesized types (inferred from terms). A similar distinction is achieved in our algorithm by manipulating type variables [14]. Also, their information flow is from functions to arguments, which is fundamentally different from the application mode.

The system of *tridirectional* type checking [15] is based on bi-directional type checking and has a rich set of property types including intersections, unions and quantified dependent types, but without parametric polymorphism. Tridirectional type checking has a new direction for supporting type checking unions and existential quantification. Their third mode is basically unrelated to our application mode, which propagates information from outer applications.

Greedy bi-directional polymorphism [13] adopts a greedy idea from Cardelli [4] on bi-directional type checking with higher ranked types, where the type variables in instantiations are determined by the first constraint. In this way, they support some uses of impredicative polymorphism. However, the greediness also makes many obvious programs rejected.

6.2 Type Inference for Higher-Ranked Types

As a reference, Fig. 5 [14,20] gives a high-level comparison between related works and our system.

Predicative Systems. Peyton Jones et al. [27] developed an approach for type inference for higher rank types using traditional bi-directional type checking based on Odersky and Läufer [24]. However in their system, in order to do instantiation on higher rank types, they are forced to have an additional type category (ρ types) as a special kind of higher rank type without top-level quantifiers. This complicates their system since they need to have additional rule sets for such types. They also combine a variant of the containment relation from Mitchell [23] for deep skolemisation in subsumption rules, which we believe is compatible with our subtyping definition.

Dunfield and Krishnaswami [14] build a simple and concise algorithm for higher ranked polymorphism based on traditional bidirectional type checking. They deal with the same language of Peyton Jones et al. [27], except they do not have *let* expressions nor generalization (though it is discussed in design variations). They have a special *application judgment* which delays instantiation until the expression is applied to some argument. As with application mode, this avoids the additional category of types. Unlike their work, our work supports generalization and HM-style *let* expressions. Moreover the use of an application mode in our work introduces several differences as to when and where annotations are needed (see Sect. 2.4 for related discussion).

Impredicative Systems. ML^F [18,19,32] generalizes ML with first-class polymorphism. ML^F introduces a new type of bounded quantification (either rigid or flexible) for polymorphic types so that instantiation of polymorphic bindings is delayed until a principal type is found. The HML system [20] is proposed as a simplification and restriction of ML^F. HML only uses flexible types, which simplifies the type inference algorithm, but retains many interesting properties and features.

The FPH system [35] introduces boxy monotypes into System F types. One critique of boxy type inference is that the impredicativity is deeply hidden in the algorithmic type inference rules, which makes it hard to understand the interaction between its predicative constraints and impredicative instantiations [31].

System	Types	Impred	Let	Annotations
ML^F	flexible and rigid	yes	yes	on polymorphically used parameters
HML	flexible F-types	yes	yes	on polymorphic parameters
FPH	boxy F-types	yes	yes	on polymorphic parameters and some let bindings with higher-ranked types
Peyton Jones et al. (2007)	F-types	no	yes	on polymorphic parameters
Dunfield et al. (2013)	F-types	no	no	on polymorphic parameters
this paper	F-types	no	sugar	on polymorphic parameters that are not applied

Fig. 5. Comparison of higher-ranked type inference systems.

6.3 Tracking Type Equalities

Tracking type equalities is useful in various situations. Here we discuss specifically two related cases where tracking equalities plays an important role.

Type Equalities in Type Checking. Tracking type equalities is one essential part for type checking algorithms involving Generalized Algebraic Data Types (GADTs) [6,26,33]. For example, Peyton Jones et al. [26] propose a type inference algorithm based on unification for GADTs, where type equalities only apply to user-specified types. However, reasoning about type equalities in GADTs is essentially different from the approach in Sect. 4: type equalities are introduced by pattern matches in GADTs, while they are introduced through type applications in our system. Also, type equalities in GADTs are local, in the sense different branches in pattern matches have different type equalities for the same type variable. In our system, a type equality is introduced globally and is never changed. However, we believe that they can be made compatible by distinguishing different kinds of equalities.

Equalities in Declarations. In systems supporting dependent types, type equalities can be introduced by declarations. In the variant of pure type systems proposed by Severi and Poll [34], expressions $x = a : A$ **in** b generate an equality $x = a : A$ in the typing context, which can be fetched later through δ-reduction. However, δ-reduction rules require careful design, and the conversion rule of δ-reduction makes the type system non-deterministic. One potential usage of the application mode is to help reduce the complexity for introducing declarations in those type systems, as briefly discussed in Sect. 5.3.

7 Conclusion

We proposed a variant of bi-directional type checking with a new *application mode*, where type information flows from arguments to functions in applications. The application mode is essentially a generalization of the inference mode, can therefore work naturally with inference mode, and avoid the rule duplication

that is often needed in traditional bi-directional type checking. The application mode can also be combined with the checked mode, but this often does not add expressiveness. Compared to traditional bi-directional type checking, the application mode opens a new path to the design of type inference/checking.

We have adopted the application mode in two type systems. Those two systems enjoy many interesting properties and features. However as bi-directional type checking can be applied to many type systems, we believe application mode is applicable to various type systems. One obvious potential future work is to investigate more systems where the application mode brings benefits. This includes systems with subtyping, intersection types [8,30], static overloading, or dependent types.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816.

References

1. Abel, A.: Termination checking with types. RAIRO-Theor. Inform. Appl. **38**(4), 277–319 (2004)
2. Abel, A., Coquand, T., Dybjer, P.: Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 29–56. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_4
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: A bi-directional refinement algorithm for the calculus of (co) inductive constructions. Log. Meth. Comput. Sci. **8**, 1–49 (2012)
4. Cardelli, L.: An implementation of FSub. Technical report, Research report 97. Digital Equipment Corporation Systems Research Center (1993)
5. Chen, G.: Coercive subtyping for the calculus of constructions. In: POPL 2003 (2003)
6. Cheney, J., Hinze, R.: First-class phantom types. Technical Report CUCIS TR2003-1901. Cornell University (2003)
7. Chlipala, A., Petersen, L., Harper, R.: Strict bidirectional type checking. In: International Workshop on Types in Languages Design and Implementation (2005)
8. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Math. Log. Q. **27**(2–6), 45–58 (1981)
9. Coq Development Team: The Coq proof assistant, Documentation, system download (2015)
10. Coquand, T.: An algorithm for type-checking dependent types. Sci. Comput. Program. **26**(1–3), 167–177 (1996)
11. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL 1982 (1982)
12. Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP 2000 (2000)
13. Dunfield, J.: Greedy bidirectional polymorphism. In: Workshop on ML (2009)
14. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: ICFP 2013 (2013)

15. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL 2004 (2004)
16. Girard, J.-Y.: The system F of variable types, fifteen years later. Theor. Comput. Sci. **45**, 159–192 (1986)
17. Hindley, J.R.: The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969)
18. Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of system F. In: ICFP 2003 (2003)
19. Le Botlan, D., Rémy, D.: Recasting MLF. Inform. Comput. **207**(6), 726–785 (2009)
20. Leijen, D.: Flexible types: robust type inference for first-class polymorphism. In: POPL 2009 (2009)
21. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently typed lambda calculus. Fundamenta informaticae **102**(2), 177–207 (2010)
22. Lovas, W.: Refinement types for logical frameworks. Ph.D. thesis, Carnegie Mellon University (2010). AAI3456011
23. Mitchell, J.C.: Polymorphic type inference and containment. Inform. Comput. **76**(2–3), 211–249 (1988)
24. Odersky, M., Läufer, K.: Putting type annotations to work. In: POPL 1996 (1996)
25. Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. In: POPL 2001 (2001)
26. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: ICFP 2006 (2006)
27. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. J. Funct. Program. **17**(01), 1–82 (2007)
28. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: POPL 2008 (2008)
29. Pierce, B.C., Turner, D.N.: Local type inference. TOPLAS **22**(1), 1–44 (2000)
30. Pottinger, G.: A type assignment for the strongly normalizable λ-terms. In: To HB Curry: essays on combinatory logic, lambda calculus and formalism. pp. 561–577 (1980)
31. Rémy, D.: Simple, partial type-inference for system F based on type-containment. In: ICFP 2005 (2005)
32. Rémy, D., Yakobowski, B.: From ML to MLF: graphic type constraints with efficient type inference. In: ICFP 2008 (2008)
33. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for gadts. In: ICFP 2009 (2009)
34. Severi, P., Poll, E.: Pure type systems with definitions. In: Nerode, A., Matiyasevich, Y.V. (eds.) LFCS 1994. LNCS, vol. 813, pp. 316–328. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58140-5_30
35. Vytiniotis, D., Weirich, S., Peyton Jones, S.: FPH: First-class polymorphism for haskell. In: ICFP 2008 (2008)
36. Wells, J.B.: Typability and type checking in system F are equivalent and undecidable. Ann. Pure Appl. Log. **98**(1–3), 111–156 (1999)
37. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL 1999 (1999)