Let Arguments Go First

Full version, including supplementary materials

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

The University of 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, 15, 14], systems with dependent types [38, 10, 2, 21, 3], and various other works [1, 13, 28, 7, 22]. Furthermore, bi-directional type checking has also been combined with HM-style techniques for providing type inference in the presence of higher-ranked types [27, 14].

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:

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

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 `(λf:Int → Int. f) (λy. 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 `A → 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.

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

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)`.

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

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 Section 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 Section 1:

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

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 Section 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: 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

[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.

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:` $\forall$`a. (a` $\rightarrow$ `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 $\mathsf{Int} \vdash \forall a.a \rightarrow a \leq B$ and we can determine that $a = \mathsf{Int}$ and $B = \mathsf{Int} \rightarrow \mathsf{Int}$. 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:

let `x =` e_1 **in** e_2 $\rightsquigarrow$ `(`λ`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 [37]. Therefore, several partial type inference algorithms, exploiting additional type annotations, have been proposed in the past instead [25, 15, 31, 27].

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

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

With this approach, nested lets can lead to types which are *more general* than HM. For example, **let** `s` = λ`x. x` **in let** `t` = λ`y. s` **in** `e`. The type of `s` is $\forall$`a. a` $\rightarrow$ `a` after generalization. Because `t` returns `s` as a result, we might expect `t:` $\forall$`b. b` $\rightarrow$ `(`$\forall$`a. a` $\rightarrow$ `a)`, which is what our system will return. However, HM will return type `t:` $\forall$`b.` $\forall$`a. b` $\rightarrow$ `(a` $\rightarrow$ `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 Section 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 Section 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 `(`λ`f : (`$\forall$`a. a` $\rightarrow$ `a)` $\rightarrow$ `(Int, Char). f) (`λ`g. (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 Section 3.4. Finally an algorithmic type system is discussed in Section 3.5.

3.1 Syntax

The syntax of the language is:

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

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 Figure 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 Section 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 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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 1. Syntax-directed typing and subtyping.

$x : A$ 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):

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

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 $(\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2)$ to $((\lambda x.\ e_2)\ e_1)$, we have the following derivation:

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

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 Figure 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 $\Gamma \shortmid \Psi \vdash e \;\Rightarrow\; A$, then for some A', we have $A = \Psi \to A'$.*

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

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 Figure 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 [27, 14], application subtyping provides a way to solve instantiation more *locally*, since it does not mutually depend on typing.

Back to the rules in Figure 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 Figure 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 Figure 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 Sections 3.2 and 3.3.

4.1 Syntax

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

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

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

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.

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 Figure 2.

Well-formedness. The type well-formedness under typing contexts is given in Figure 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.

4.2 Type System

Typing Judgments. From Lemma 1 and Lemma 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 Figure 1:

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

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

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

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

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

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

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

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

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

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

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

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 Section 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 $\Gamma \mid \Psi \vdash e \;\Rightarrow\; A$ 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 Figure 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 Figure 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 Section 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:

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

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.

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

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 Section 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:

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

However, in polymorphic type systems, we need to take the subsumption rule into consideration. For example, in the expression $(\lambda x : (\forall a.(a, b))$. **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:

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

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

An extended version of the calculus presented in Section 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 Section 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 [38, 10, 2, 21, 3]. 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 $\mathbf{let}\, a = \mathsf{Int}\, \mathbf{in}\, \lambda x : a.\ x + 1$ 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 Section 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.

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

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.

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.

6.2 Type Inference for Higher-Ranked Types

As a reference, Figure 5 [20, 14] 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 Section 2.4 for related discussion).

Impredicative Systems. ML^F [18, 32, 19] 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 [36] 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].

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 Section 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 Section 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 [30, 8], 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.

Bibliography

[1] Andreas Abel. Termination checking with types. *RAIRO-Theoretical Informatics and Applications*, 38(4):277–319, 2004.

[2] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory. In *International Conference on Mathematics of Program Construction*, 2008.

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co) inductive constructions. *Logical Methods in Computer Science*, 8:1–49, 2012.

[4] Luca Cardelli. An implementation of fsub. Technical report, Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[5] Gang Chen. Coercive subtyping for the calculus of constructions. POPL '03, 2003.

[6] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[7] Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *International Workshop on Types in Languages Design and Implementation*, 2005.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6): 45–58, 1981.

[9] Coq Development Team. The *Coq* proof assistant, 2015. Documentation, system download.

[10] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[11] Luis Damas and Robin Milner. Principal type-schemes for functional programs. POPL '82, 1982.

[12] Rowan Davies and Frank Pfenning. Intersection types and computational effects. ICFP '00, 2000.

[13] Joshua Dunfield. Greedy bidirectional polymorphism. In *Workshop on ML*, 2009.

[14] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. ICFP '13, 2013.

[15] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. POPL '04, 2004.

[16] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.

[17] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[18] Didier Le Botlan and Didier Rémy. Mlf: Raising ml to the power of system f. ICFP '03, 2003.

[19] Didier Le Botlan and Didier Rémy. Recasting mlf. *Information and Computation*, 207(6):726–785, 2009.

[20] Daan Leijen. Flexible types: Robust type inference for first-class polymorphism. POPL '09, 2009.

[21] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

[22] William Lovas. *Refinement Types for Logical Frameworks*. PhD thesis, Carnegie Mellon University, 2010. AAI3456011.

[23] John C Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.

[24] Martin Odersky and Konstantin Läufer. Putting type annotations to work. POPL '96, 1996.

[25] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. POPL '01, 2001.

[26] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. ICFP '06, 2006.

[27] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(01):1–82, 2007.

[28] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. POPL '08, 2008.

[29] Benjamin C Pierce and David N Turner. Local type inference. *TOPLAS*, 22(1):1–44, 2000.

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

[31] Didier Rémy. Simple, partial type-inference for system f based on type-containment. ICFP '05, 2005.

[32] Didier Rémy and Boris Yakobowski. From ml to mlf: Graphic type constraints with efficient type inference. ICFP '08, 2008.

[33] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. ICFP '09, 2009.

[34] Paula Severi and Erik Poll. Pure type systems with definitions. *Logical Foundations of Computer Science*, pages 316–328, 1994.

[35] Dimitrios Vytiniotis, Stephanie C Weirich, and Simon Peyton Jones. Practical type inference for arbitrary-rank types: Technical appendix. *Technical Reports (CIS)*, page 58, 2005.

[36] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Fph: First-class polymorphism for haskell. ICFP '08, 2008.

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

[38] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. POPL '99, 1999.

A Appendix Overview

This sections gives an overview introduction of the following several appendixes.

A.1 Appendix B

This appendix is a full description of the translation process shown in Section 3.4.

A.2 Appendix C

This appendix gives more definitions about Section 4, including a full definition of well-formedness of contexts, and operational semantics.

A.3 Appendix D

Appendix D gives two proofs that complement the Coq code, where two lemmas about generalization are proved.

Section D.1 This section defines the notations that are used in proofs, including arrows on application contexts, and subtyping on typing contexts and application contexts.

Section D.2 This section gives lemmas that are used in proofs. Those lemmas are all proved in Coq code. For the reference, here is the lists of all the lemmas:

1. Lemma 21 Helper Lemmas (`lemma:denv_sub_binds` in DEnvSub.v)
2. Lemma 22 Helper Lemmas (`lemma:dsub_stack_typ` in DeclSound.v)
3. Lemma 23 Helper Lemmas (`lemma:dsub_weaker_stack` in DeclSound.v)
4. Lemma 24 Helper Lemmas (`lemma:dsub_stack_to_plain` in DeclSound.v)
5. Lemma 25 Helper Lemmas (`lemma:dsub_plain_to_stack` in DeclSound.v)
6. Lemma 26 Helper Lemmas (`lemma:sub_trans` in DeclSound.v)
7. Lemma 27 Helper Lemmas (`lemma:dsub_subst_stack_var` in DeclSound.v)
8. Lemma 28 Helper Lemmas (`lemma:dgen_sub` in DeclSound.v)
9. Lemma 29 Helper Lemmas (`lemma:dtyping_stack_typ` in DeclSound.v)
10. Lemma 30 Helper Lemmas (`lemma:dtyping_subst_tvar` in DeclSound.v)
11. Lemma 31 Helper Lemmas (`lemma:dgen_subst_tvar` in DeclSound.v)

Section D.3 The section includes proofs for two lemmas that cannot easily be proved through Coq code.

1. Lemma 32 Proof (`lemma:dgen_exists` in HMPreserve.v)
2. Lemma 33 Proof (`lemma:dtyping_size_weaken_helper` in HMPreserve.v)

Section D.4 This section extends the original proofs to account for pairs.

A.4 Appendix E

Appendix E gives the algorithmic system and proves its soundness and completeness with respect to the specification.

Section E.1 This section introduces the name supply N and substitution S, which are used as input and output in the algorithms.

Section E.2 The algorithmic system is given in this section, which extends the specification to give deterministic results. The components of the algorithm include:

1. Section E.2 Unification
2. Section E.2 Arrow Unification
3. Section E.2 Subtyping
4. Section E.2 Typing

Section E.3 The main proofs about the algorithmic system is in this section, which contains all the auxiliary lemmas (Section E.3 Auxiliary), and the proof of soundness (Section E.3 Soundness) and completeness (Section E.3 Completeness).

- Theorem 3 Soundness
  - Lemma 37 Soundness
  - Lemma 38 Soundness
  - Lemma 39 Soundness
  - Lemma 40 Soundness
  - Lemma 41 Soundness
- Theorem 4 Completeness
  - Lemma 47 Completeness
  - Lemma 48 Completeness
  - Lemma 49 Completeness
  - Lemma 50 Completeness

Section E.4 This section extends the algorithmic type system and all the proofs to account for the addition of pairs.

$$\boxed{\Gamma \vdash^F e : B}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash^F x : A}\ \text{F-Var} \qquad \frac{}{\Gamma \vdash^F n : \mathsf{Int}}\ \text{F-Int} \qquad \frac{\Gamma, x : A \vdash^F e : B}{\Gamma \vdash^F \lambda x : A.\ e : A \to B}\ \text{F-LamAnn}$$

$$\frac{\Gamma \vdash^F e : \sigma \qquad a \notin ftv(\Gamma)}{\Gamma \vdash^F \Lambda a.e : \forall a.\sigma}\ \text{F-BLam} \qquad \frac{\Gamma \vdash^F e_1 : A \to B \qquad \Gamma \vdash^F e_2 : A}{\Gamma \vdash^F e_1\ e_2 : B}\ \text{F-App}$$

$$\frac{\Gamma \vdash^F e_1 : \forall a.A}{\Gamma \vdash^F e_1\ B : A[\![a \mapsto B]\!]}\ \text{F-TApp}$$

Fig. 6. System F Typing.

B Translation

This section discusses the type-directed translation of the source language presented in Section 3 into a variant of System F [16] that is also used in Peyton Jones et al. [27]. The translation is shown to be coherent and type safe. The later result implies the type-safety of the source language. To prove coherency, we need to decide when two translated terms are the same using *η-id equality*, and show that the translation is unique up to this definition.

B.1 Target Language

Our target language is one variant of System F, which is the same as in the work by Peyton Jones et al. [27]. The syntax is as follows:

$$\begin{array}{lrl}
\text{Expressions} & s, f & ::= x \mid n \mid \lambda x : A.\ s \mid \Lambda a.s \mid s_1\ s_2 \mid s_1\ A \\
\text{Types} & A, B, C, D & ::= a \mid A \to B \mid \forall a.A \mid \mathsf{Int} \\
\text{Typing Contexts} & \Gamma & ::= \varnothing \mid \Gamma, x : A
\end{array}$$

Expressions include variables x, integers n, annotated abstractions $\lambda x : A.\ s$, type-level abstractions $\Lambda a.s$, and $s_1\ s_2$ for term application, and $s_1\ A$ for type application. The types and the typing contexts are the same as our system, where typing contexts does not track type variables. In translation, we use f to refer to the coercion function produced by subtyping translation, and s to refer to the translated term in System F.

For reference, Figure 6 gives the typing rules for System F.

B.2 Subtyping Coercions

The type-directed translation of subtyping is shown in Figure 7. The translation follows the subtyping relations from Figure 1, but adds a new component. The judgment $A <: B \rightsquigarrow f$ is read as: if $A <: B$ holds, it can be translated to a

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

$$\frac{}{\mathsf{Int} \;<:\; \mathsf{Int} \rightsquigarrow \lambda x : \mathsf{Int}.\ x}\ \text{S-Int} \qquad \frac{}{a \;<:\; a \rightsquigarrow \lambda x : a.\ x}\ \text{S-Var}$$

$$\frac{A \;<:\; B \rightsquigarrow f}{A \;<:\; \forall a.B \rightsquigarrow \lambda x : A.\ \Lambda a.f\ x}\ \text{S-ForallR}$$

$$\frac{A[\![a \mapsto \tau]\!] \;<:\; B \rightsquigarrow f}{\forall a.A \;<:\; B \rightsquigarrow \lambda x : \forall a.A.\ f\ (x\ \tau)}\ \text{S-ForallL}$$

$$\frac{C \;<:\; A \rightsquigarrow f_1 \qquad B \;<:\; D \rightsquigarrow f_2}{A \to B \;<:\; C \to D \rightsquigarrow \lambda x : A \to B.\ \lambda y : C.\ f_2\ (x\ (f_1\ y))}\ \text{S-Fun}$$

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

$$\frac{}{\varnothing \vdash A \;<:\; A \rightsquigarrow \lambda x : A.\ x}\ \text{S-Empty}$$

$$\frac{\Psi, C \vdash A[\![a \mapsto \tau]\!] \;<:\; B \rightsquigarrow f}{\Psi, C \vdash \forall a.A \;<:\; B \rightsquigarrow \lambda x : \forall a.A.\ f\ (x\ \tau)}\ \text{S-ForallL2}$$

$$\frac{C \;<:\; A \rightsquigarrow f_1 \qquad \Psi \vdash B \;<:\; D \rightsquigarrow f_2}{\Psi, C \vdash A \to B \;<:\; C \to D \rightsquigarrow \lambda x : A \to B.\ \lambda y : C.\ f_2\ (x\ (f_1\ y))}\ \text{S-Fun2}$$

Fig. 7. Subtyping translation to System F.

coercion function f in System F. The coercion function produced by subtyping is used to transform values from one type to another. So, in theory, we should have $\emptyset \vdash^F f : A \to B$.

Rules S-Int and S-Var produce identity functions, since the source type and target type are the same. Rule S-ForallR uses the coercion f and, in order to produce a polymorphic type, we add one type abstraction to turn it into a coercion of type $A \to \forall a.B$. In rule S-ForallL, the input argument is a polymorphic type, so we feed the type τ to it and apply the coercion function f from the precondition. In S-Fun, the coercion function f_1 of type $C \to A$ is applied to y to get a value of type A. Then the resulting value becomes an argument to x, and a value of type B is obtained. Finally we apply f_2 to the value of type B, so that a value of type D is computed.

The second part of the subtyping translation deals with coercions generated by subtyping with application contexts. The judgment $\Psi \vdash A \;<:\; B \rightsquigarrow f$ is read as: if $\Psi \vdash A \;<:\; B$ holds, it can be translated to a coercion function f in System F. If we compare two parts, we can see application contexts play no role in the generation of the coercion. Notice the similarity between S-Var and S-Empty, between S-ForallR and S-ForallR2, and between S-Fun and S-Fun2. We therefore omit more explanations, since the translation is concise enough.

B.3 Type-Directed Translation of Typing

The type directed translation of typing is shown in the Figure 8, which extends the rules in Figure 1. The judgment $\Gamma \mid \Psi \vdash e \Rightarrow A \rightsquigarrow s$ is read as: if $\Gamma \mid \Psi \vdash e \Rightarrow A$ holds, it can be translated to term s in System F.

In rule T-Var, x is translated to $f\ x$, where f is the coercion function generated from subtyping. In rule T-Int, integers remain integers. Rules T-Lam and T-Lam2 work as expected. In rule T-LamAnn1, we use the translated body to create a new abstraction. Rule T-LamAnn2 applies the coercion function f to y, then feeds y to the function generated from the abstraction body. Rule T-App relies on rule T-Gen, the latter one will generate type-level abstractions on a term. Notice T-Gen now takes an additional input: the coercion s_2 resulting from the translation of e_2.

B.4 Type Safety

To show type safety, we need to show that our translation produces terms that are well typed under System F.

Lemma 16 (Soundness of Typing). *if $\Gamma \mid \Psi \vdash e \Rightarrow A \rightsquigarrow s$, then $\Gamma \vdash^F s : A$.*

The lemma relies on the properties of translation of subtyping and generalization: subtyping translation produce type-correct coercions, and generalization produce type-correct terms.

Lemma 17.

1. *If $A <: B \rightsquigarrow f$, then $\varnothing \vdash^F f : A \to B$.*
2. *If $\Psi \vdash A <: B \rightsquigarrow f$, then $\varnothing \vdash^F f : A \to B$.*
3. *if $\Gamma \vdash^F s_1 : A$, and $\Gamma_{gen}(A, s_1) = B \rightsquigarrow s_2$, then $\Gamma \vdash^F s_2 : B$.*

B.5 Coherence

One problem with the translation is that there are multiple targets corresponding to one expression. This is because in original system there are multiple choices when instantiating a polymorphic type, or guessing the annotation for unannotated lambda abstraction (rule T-Lam2). For each choice, the corresponding target will be different. For example, expression $\lambda x.\ x$ can be type checked with $\mathsf{Int} \to \mathsf{Int}$, or $a \to a$, and the corresponding targets are $\lambda x : \mathsf{Int}.\ x$, and $\lambda x : a.\ x$.

Therefore, in order to prove the translation is coherent, we turn to prove that all the translations have the same operational semantics. There are two steps towards the goal: type erasure, and considering η expansion and identity functions.

$$\boxed{\Gamma \mid \Psi \vdash e \;\Rightarrow\; A \rightsquigarrow s}$$

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

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

$$\frac{\Gamma, x : \tau \vdash e \;\Rightarrow\; B \rightsquigarrow s}{\Gamma \vdash \lambda x.\ e \;\Rightarrow\; \tau \to B \rightsquigarrow \lambda x : \tau.\ s}\ \text{T-Lam2}$$

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

$$\frac{C \;<:\; A \rightsquigarrow f \qquad \Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B \rightsquigarrow s}{\Gamma \mid \Psi, C \vdash \lambda x : A.\ e \;\Rightarrow\; C \to B \rightsquigarrow \lambda y : C.\ (\lambda x : A.\ s)\ (f\ y)}\ \text{T-LamAnn2}$$

$$\frac{\bar{a} = ftv(A) - ftv(\Gamma)}{\Gamma_{gen}(A, s) = \forall \bar{a}.A \rightsquigarrow \Lambda \bar{a}.s}\ \text{T-Gen}$$

$$\frac{\Gamma \vdash e_2 \;\Rightarrow\; A \rightsquigarrow s_2 \qquad \Gamma_{gen}(A, s_2) = B \rightsquigarrow s_3 \qquad \Gamma \mid \Psi, B \vdash e_1 \;\Rightarrow\; B \to C \rightsquigarrow s_1}{\Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; C \rightsquigarrow s_1\ s_3}\ \text{T-App}$$

Fig. 8. Typing translation to System F.

$$\begin{array}{lll} |x| = x & & |\Lambda a.s| = |s| \\ |n| = n & & |s_1\ s_2| = |s_1|\ |s_2| \\ |\lambda x : A.\ s| = \lambda x.\ |s| & & |s_1\ A| = |s_1| \end{array}$$

Fig. 9. Type Erasure of System F.

Type Erasure. Since type information is useless after type-checking, we erase the type information of the targets for comparison. The erasure process is defined in Figure 9. The erasure process is standard, where we erase the type annotation in abstractions, and remove type abstractions and type applications. The calculus after erasure is the untyped lambda calculus.

η-id Equality. However, even if we have type erasure, multiple targets for one expression can still be syntactically different. The problem is that we can insert more coercion functions in one translation than another, since an expression can have a more polymorphic type in one derivation than another one. So we need a more refined definition of equality instead of syntactic equality.

We use a similar definition of η-id equality as in Chen [5], shown in Figure 10. In η-id equality, two expressions are regarded as equivalent if they can turn into the same expression through η-reduction or removal of redundant identity functions. η-id equality is reflexive, symmetrical, and transitive. As a small ex-

ample illustrating η-id equality we can show that $\lambda x.\ (\lambda x.\ x)\ f\ x \rightsquigarrow_{\eta id} f$ holds by Eta-reduce and Id-reduce.

$$\boxed{e_1 \rightsquigarrow_{\eta id} e_2}$$

$$\frac{x \notin ftv(e)}{\lambda x.\ e\ x \rightsquigarrow_{\eta id} e}\ \textsc{Eta-Reduce} \qquad \frac{}{(\lambda x.\ x)\ e \rightsquigarrow_{\eta id} e}\ \textsc{Id-Reduce}$$

$$\frac{e_1 \rightsquigarrow_{\eta id} e_1' \qquad e_2 \rightsquigarrow_{\eta id} e_2'}{e_1\ e_2 \rightsquigarrow_{\eta id} e_1'\ e_2'}\ \textsc{Eta-App} \qquad \frac{e \rightsquigarrow_{\eta id} e'}{\lambda x.\ e \rightsquigarrow_{\eta id} \lambda x.\ e'}\ \textsc{Eta-Abs}$$

$$\frac{}{e \rightsquigarrow_{\eta id} e}\ \textsc{Eta-Refl} \qquad \frac{e \rightsquigarrow_{\eta id} e'}{e' \rightsquigarrow_{\eta id} e}\ \textsc{Eta-Symm} \qquad \frac{e_1 \rightsquigarrow_{\eta id} e_2 \qquad e_2 \rightsquigarrow_{\eta id} e_3}{e_1 \rightsquigarrow_{\eta id} e_3}\ \textsc{Eta-Tran}$$

Fig. 10. ηid equality in Erasure Target.

Now we first prove that the translation of subtyping is always η-id equivalent to identity function.

Lemma 18. *if* $\Gamma \vdash A\ <:\ B \rightsquigarrow f$, *then* $|f| \rightsquigarrow_{\eta id} (\lambda x.\ x)$.

Moreover generalizations for one expression result in equivalent targets:

Lemma 19. *if* $\Gamma_{gen}(A, s_1) = B \rightsquigarrow s_2$, *then* $|s_1| = |s_2|$.

With these two lemmas, we can now prove our translation actually generates only "one" target:

Lemma 20 (Coherence). *If* $\Gamma_1 \shortmid \Psi_1 \vdash e\ \Rightarrow\ A \rightsquigarrow s_1$, *and* $\Gamma_2 \shortmid \Psi_2 \vdash e\ \Rightarrow\ B \rightsquigarrow s_2$, *then* $|s_1| \rightsquigarrow_{\eta id} |s_2|$.

$$\boxed{\Gamma \ ctx}$$

$$\frac{}{\varnothing \ ctx}\ \text{WC-Empty} \qquad \frac{\Gamma \ ctx \qquad x \notin \Gamma \qquad \Gamma \vdash A}{\Gamma, x : A \ ctx}\ \text{WC-Var}$$

$$\frac{\Gamma \ ctx \qquad a \notin \Gamma}{\Gamma, a \ ctx}\ \text{WC-TVar} \qquad \frac{\Gamma \ ctx \qquad a \notin \Gamma \qquad \Gamma \vdash A}{\Gamma, a = A \ ctx}\ \text{WC-TEq}$$

$$\boxed{\Gamma \vdash \Psi}$$

$$\frac{}{\Gamma \vdash \varnothing}\ \text{WA-Empty} \qquad \frac{\Gamma \vdash \Psi \qquad \Gamma \vdash A}{\Gamma \vdash \Psi, A}\ \text{WA-Typ} \qquad \frac{\Gamma \vdash \Psi \qquad \Gamma \vdash A}{\Gamma \vdash \Psi, [A]}\ \text{WA-TVar}$$

Fig. 11. Well-formedness of typing contexts and application contexts.

$$\boxed{e_1 \longrightarrow e_2}$$

$$\frac{}{(\lambda x.\ e_1)\ e_2 \longrightarrow e_1[\![x \mapsto e_2]\!]}\ \text{SF-Abs1} \qquad \frac{}{(\lambda x : A.\ e_1)\ e_2 \longrightarrow e_1[\![x \mapsto e_2]\!]}\ \text{SF-Abs2}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{SF-App} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ [A] \longrightarrow e_1'\ [A]}\ \text{SF-TApp}$$

$$\frac{}{(\Lambda a.e)\ [A] \longrightarrow e_1[\![a \mapsto A]\!]}\ \text{SF-TAbs}$$

Fig. 12. Operational Semantics.

C Definitions for a Variant of System F

This section gives the definitions for the system in Section 4.

C.1 Well-formedness

The well-formedness of a typing context is given at the top of Figure 11. WC-Empty states an empty context is always well-formed. WC-Var requires the variable is fresh and the type is well-formed under current typing context. Similarly, WC-TVar requires the type variable is fresh. WC-TEq is for type equations and requires A is well-formed.

The well-formedness of an application context under a typing context is given at the bottom of Figure 11. In WA-Empty, an empty application context is always well-formed. Both in WA-Typ and WA-TVar, the type is required to be well-formed.

C.2 Operational Semantics

The operational semantics is given in Figure 12. SF-Abs1 and SF-Abs2 do beta-reduction. SF-App takes one step in the application. SF-TApp takes one step in type application. SF-TAbs does beta-reduction on types.

The operational semantics is call-by-name. The choice of evaluation strategy is only a matter of taste.

D Complementary Proof for Coq

Coq is proficient at list operation, but not at set. This is also because set is a more subtle data structure than list: splitting a lists into two part is easy in Coq, but splitting a set is not; equivalence between lists is easy to prove, but between set is not.

Due to our definition of generalization, which heavily relies on set operations, the definition of generalization in Coq is in a more inductive way:

$$\frac{ftv(A) - ftv(\Gamma) = \varnothing}{\Gamma_{gen}(A, s) = A \rightsquigarrow s}\ \text{T-Gen1} \qquad \frac{a \in ftv(A) - ftv(\Gamma) \qquad \Gamma_{gen}(\forall a.A, \Lambda a.s) = B \rightsquigarrow s_2}{\Gamma_{gen}(A, s) = B \rightsquigarrow s_2}\ \text{T-Gen2}$$

With this definition, we are capable of proving lots of lemmas. However, there are still two lemmas related to generalization we are not able to prove even under this definition. So here we give the hand-written proofs in a mathematical style.

D.1 Notation

For simplification and clarity, we use some handy notations to help present the proof.

For a application context Ψ contains $A_n, ..., A_2, A_1$, and a type B, $\Psi \to B$ means make the arrow type $A_1 \to A_2 \to ... \to A_n \to B$.

We define *subtyping* relation on typing context and application context:

- $\Gamma_1 <: \Gamma_2$: means Γ_1 and Γ_2 have the same domain, and all the types for the same variable in Γ_1 is more general than Γ_2.
- $\Psi_1 <: \Psi_2$: means Ψ_1 and Ψ_2 have the same length, and all the types in the same position in Ψ_1 is more general than Ψ_2.

D.2 Helper Lemmas

During the proof, it will refers to some lemmas already proved in Coq. Here we list the lemmas for reference. They can all be found in our Coq code.

Lemma 21 (Binds Environment Weakening). *if* $x : A \in \Gamma_1$, *and* $\Gamma_2 <: \Gamma_1$, *then* $\exists B$, *that* $x : B \in \Gamma_2$, *and* $B <: A$.

Lemma 22 (Subtyping Stack Form). *if* $\Psi \vdash A <: B$, *then* $\exists C, B = \Psi \to C$

Lemma 23 (Subtyping Stack Weakening). *if* $\Psi_1 \vdash A <: \Psi_1 \to B$, *and* $\Psi_2 <: \Psi_1$, *then* $\exists C$, *that* $\Psi_2 \vdash A <: \Psi_2 \to C$, *and* $C <: B$.

Lemma 24 (Subtyping Remove Stack). *if* $\Psi \vdash A <: B$, *then* $A <: B$.

Lemma 25 (Subtyping Add Stack). *if* $A <: \Psi \to B$, *then* $\exists C$, *that* $\Psi \vdash A <: \Psi \to C$, *and* $C <: B$

Lemma 26 (Subtyping Transitivity). *if* $A <: B$, *and* $B <: C$, *then* $A <: C$.

Lemma 27 (Subtyping Substitution). *if* $\Psi \vdash A <: B$, *then* $\Psi[\![x \mapsto \tau]\!] \vdash A[\![x \mapsto \tau]\!] <: B[\![x \mapsto \tau]\!]$

Lemma 28 (Generalization Subtyping). *if* $\Gamma_{gen}(A) = B$, *then* $B <: A$.

Lemma 29 (Typing Stack Form). *if* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$, *then* $\exists B$, *that* $A = \Psi \to B$.

Lemma 30 (Typing Substitution). *if* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$, *then* $\exists L$, *that* $\forall y$, *if* $y \notin L$, *then* $\Gamma[\![x \mapsto y]\!] \shortmid \Psi[\![x \mapsto y]\!] \vdash e \Rightarrow T[\![x \mapsto y]\!]$.

This lemma is actually a weaker version of typing substitution, again due to the complex of set operation related to generalization. But it is sufficient for the proof's purpose.

Lemma 31 (Generalization Substitution). *if* $\Gamma_{gen}(A) = B$, *and* $b \notin ftv(\Gamma, A)$, *then* $\overline{\Gamma[\![a \mapsto b]\!](A[\![a \mapsto b]\!])} = B[\![a \mapsto b]\!]$

D.3 Proof

Lemma 32 (Generalization Existence). $\forall E, t$, *we can generalize* t *under* E *to* $\forall \bar{a}.t$, *where* $\bar{a} = ftv(t) - ftv(E)$.

This lemma holds trivially from the principle of generalization. □

Lemma 33 (Generalization Weaken). *if* $\Gamma_1 \shortmid \Psi_1 \vdash e \Rightarrow \Psi_1 \to A_1$, *and* $\Gamma_2 <: \Gamma_1$, *and* $\Psi_2 <: \Psi_1$, *and* $\overline{\Gamma_1(A_1)} = A_2$, *and* $ftv(\Psi_1) \subseteq ftv(\Gamma_1)$, *and* $ftv(\Psi_2) \subseteq ftv(\Gamma_2)$, *then* $\exists B_1, B_2$, *that* $\Gamma_2 \shortmid \Psi_2 \vdash e \Rightarrow \Psi_2 \to B_1$, *and* $\overline{\Gamma_2(B_1)} = B_2$, *and* $B_2 <: A_2$.

We induction on the size of derivation, and then case analysis on the last rule used in the derivation.

– Case VAR. According to the typing rule of variable and hypothesis, we have

$$e = x \tag{1}$$
$$x : A \in \Gamma_1 \tag{2}$$
$$\Psi_1 \vdash A <: \Psi_1 \to A_1 \tag{3}$$
$$A_2 = \forall \bar{a}.A_1 \tag{4}$$

Where $\bar{a} = ftv(A_1) - ftv(\Gamma_1)$. Because we have $ftv(\Psi_1) \subseteq ftv(\Gamma_1)$, so $\bar{a} \notin \Psi_1$. Also, because A is the type for bound variable x in Γ, so $\bar{a} \notin ftv(A)$.
By substituting $\bar{a}$ to some $\bar{b}$, where $\bar{b} \notin ftv(E, F)$, with lemma subtyping substitution(27), we get

$$\Psi_1 \vdash A <: \Psi_1 \to A_1[\![\bar{a} \mapsto \bar{b}]\!] \tag{5}$$

where Ψ_1 and A stays the same because they do not have $\bar{a}$.
Because we have $\Gamma_2 <: \Gamma_1$, from equation 2 and lemma 21, we have a B, that

$$x : B \in F \tag{6}$$
$$B <: A \tag{7}$$

Because we have $\Psi_2 <: \Psi_1$, from equation 5 and lemma 23, we get a C_1, that

$$\Psi_2 \vdash A <: \Psi_2 \to C_1 \tag{8}$$
$$C_1 <: A_1[\![\bar{a} \mapsto \bar{b}]\!] \tag{9}$$

Apply lemma 24 to equation 8, we get

$$A <: \Psi_2 \to C_1 \tag{10}$$

From subtyping transitivity (lemma 26) and equation 7 and 10, we get

$$B <: \Psi_2 \to C_1 \tag{11}$$

Continue by applying lemma 25, we get a C_2 that

$$\Psi_2 \vdash B <: \Psi_2 \to C_2 \tag{12}$$
$$C_2 <: C_1 \tag{13}$$

According to lemma 32, we have $F_{gen}(C_2)$. So feed C_2 and $\overline{F(C_2)}$ to the conclusion, what we want is

$$\Gamma_2 \shortmid \Psi_2 \vdash e \Rightarrow \Psi_2 \to C_2 \tag{14}$$
$$\Gamma_{2gen}(C_2) = \Gamma_{2gen}(C_2) \tag{15}$$
$$\Gamma_{2gen}(C_2) <: \forall \bar{a}.A_2 \tag{16}$$

Equation 14 holds because we could apply typing rule with equation 6 and 12. Equation 15 holds trivially. So we deal with equation 16 now.
Because generalized result is more polymorphic than itself (lemma 28), so we have

$$\Gamma_{2gen}(C_2) <: C_2 \tag{17}$$

According to subtyping transitivity (lemma 26), from equation17, 13 and 9, we get

$$\Gamma_{2gen}(C_2) <: A_1[\![\bar{a} \mapsto \bar{b}]\!] \tag{18}$$

Because from the principle of generalization, $ftv(\Gamma_{2gen}(C_2)) \subseteq ftv(\Gamma_2)$, and we have $\bar{b} \notin ftv(\Gamma_2)$, so

$$\bar{b} \notin ftv(\Gamma_{2gen}(C_2)) \tag{19}$$
$$\Gamma_{2gen}(C_2) <: \forall \bar{b}.A_2[\![\bar{a} \mapsto \bar{b}]\!] \tag{20}$$

Equation 20 holds because $\bar{b}$ are themselves fresh to $\Gamma_{2gen}(C_2)$, so we could repeatedly apply rule S-ForallR and use equation 18 to get the subtyping relationship.
Then by α renaming of $\bar{b}$ to $\bar{a}$, equation 16 holds and we are done.

– Case INT. According to the typing rule of integers and hypothesis, we have

$$e = i \tag{21}$$
$$\Psi_1 = \Psi_2 = \emptyset \tag{22}$$
$$A_1 = \mathsf{Int} \tag{23}$$
$$A_2 = \Gamma_{1gen}(\mathsf{Int}) = \mathsf{Int} \tag{24}$$

Directly follows by using typing rule of integers.

– Case LAMANN. According to the typing rule and hypothesis, we have, for some C_1, C_2,

$$e = \lambda x : C_1.\ e_1 \tag{25}$$
$$\Psi_1 = \Psi_2 = \emptyset \tag{26}$$
$$A_1 = C_1 \to C_2 \tag{27}$$
$$\Gamma_1, x : C_1 \vdash e_1 \ \Rightarrow\ C_2 \tag{28}$$
$$A_2 = \Gamma_{1gen}(C_1 \to C_2) = \forall \bar{a}.C_1 \to C_2 \tag{29}$$

where $\bar{a} = ftv(C_1 \to C_2) - ftv(\Gamma_1)$. Because C_1 is a user defined type and has no free variables, so $\bar{a} = ftv(C_2) - ftv(\Gamma_1) = ftv(C_2) - ftv(\Gamma_1, C_1)$. So

$$(\Gamma_1, x : C_1)_{gen}(C_2) = \forall \bar{a}.C_2. \tag{30}$$

Apply induction hypothesis to equation 28, with $(\Gamma_2, x : C_1) <: (\Gamma_1, x : C_1)$, empty application contexts, equation 30, $\emptyset \subseteq ftv(\Gamma_1, x : C_1)$, $\emptyset \subseteq ftv(\Gamma_2, x : C_2)$, we get for some D_1,

$$\Gamma_2, x : C_1 \vdash e_1 \ \Rightarrow\ D_1 \tag{31}$$
$$(\Gamma_2, x : C_1)_{gen}(D_1) = \forall \bar{b}.D_1 \tag{32}$$
$$\forall \bar{b}.D_1 \ <:\ \forall \bar{a}.C_2 \tag{33}$$

where $\bar{b} = ftv(D_1) - ftv(\Gamma_2, x : C_1)$. Again because C_1 is a used defined type, so it contains no free type variables. So $\bar{b} = ftv(D_1) - ftv(\Gamma_2) = ftv(C_1 \to D_1) - ftv(\Gamma_2) = \bar{b}$. So according to lemma 32, we have

$$\Gamma_{2gen}(C_1 \to D_1) = \forall \bar{b}.C_1 \to D_1 \tag{34}$$

What we want is

$$\Gamma_{2gen}(C_1 \to D_1) \ <:\ \Gamma_{1gen}(C_1 \to C_2) \tag{35}$$

Namely,

$$\forall \bar{b}.C_1 \to D_1 \ <:\ \forall \bar{a}.C_1 \to C_2 \tag{36}$$

According to equation 33 and the covariant of function return type, equation 36 holds.

Finally, feed $C_1 \to D_1$ and $\forall \bar{b}.C_1 \to D_1$ to the conclusion, we could get what we want is

$$\Gamma_2 \vdash \lambda x : C_1.\ e_1 \ \Rightarrow\ C_1 \to D_1 \tag{37}$$

This holds by applying the typing rule for annotated lambda with equation 31. And the rests are already proved (equation 36). So we are done.

– Case LAMANN2. Similar to LAMANN case, with differences to deal with application context. According to the typing rule and hypothesis, we have

$$e = \lambda x : C_1.\ e_1 \tag{38}$$
$$\Psi_1 = (\Psi_1', C_2) \tag{39}$$
$$C_2 \ <:\ C_1 \tag{40}$$
$$\Psi_2 \ <:\ (\Psi_1', C_2) \tag{41}$$
$$\Gamma_1, x : C_1 \mid \Psi_1' \vdash e_1 \ \Rightarrow\ \Psi_1' \to A_1 \tag{42}$$
$$A_2 = \Gamma_{1gen}(A_1) = \forall \bar{a}.A_1 \tag{43}$$
$$ftv(\Psi_1', C_2) \subseteq ftv(\Gamma_1) \tag{44}$$

where $\bar{a} = ftv(A_1) - ftv(\Gamma_1)$. Because C_1 is a user defined type and has no free variables, we get $\bar{a} = ftv(A_1) - ftv(\Gamma_1, C_1)$. So

$$(\Gamma_1, x : C_1)_{gen}(A_1) = \forall \bar{a}.A_1. \tag{45}$$

Do inversion on equation 41, we get for some D_1 and Ψ_2',

$$\Psi_2 = \Psi_2', D_1 \tag{46}$$
$$D_1 \ <:\ C_2 \tag{47}$$
$$\Psi_2' \ <:\ \Psi_1' \tag{48}$$

From equation 44, it is easy to derive that

$$ftv(\Psi_1') \subseteq ftv(\Gamma_1, x : C_1) \tag{49}$$

We have $ftv(\Psi_2) \subseteq ftv(\Gamma_2)$, combining equation 46, we can derive

$$ftv(\Psi_2') \subseteq ftv(\Gamma_2, x : C_1) \tag{50}$$

Applying hypothesis on equation 42, $(\Gamma_2, x : C_1 \ <:\ \Gamma_1, x : C_1)$ from $(\Gamma_2 \ <:\ \Gamma_1)$, equation 45, equation 48, equation 49, equation 50, we get for some D_2,

$$\Gamma_2, x : C_1 \mid \Psi_2' \vdash e_1 \ \Rightarrow\ \Psi_2' \to D_2 \tag{51}$$
$$(\Gamma_2, x : C_1)_{gen}(D_2) = \forall \bar{b}.D_2 \tag{52}$$
$$\forall \bar{b}.D_2 \ <:\ \forall \bar{a}.A_1 \tag{53}$$

where $\bar{b} = ftv(D_2) - ftv(\Gamma_2, x : C_1)$. Again because C_1 is a used defined type, so it contains no free type variables. So $\bar{b} = ftv(D_2) - ftv(\Gamma_2)$. So according to lemma 32, we have

$$\Gamma_{2gen}(D_2) = \forall \bar{b}.D_2 \tag{54}$$

What we want is

$$\Gamma_{2gen}(D_2) \ <: \ \Gamma_{1gen}(A_1) \tag{55}$$

Namely,

$$\forall \bar{b}.D_2 \ <: \ \forall \bar{a}.A_1 \tag{56}$$

which is exactly equation 53.
Finally, feed D_2 and $\forall \bar{b}.D_2$ to the conclusion, we could get what we want is

$$\Gamma_2 \mid \Psi_2 \vdash \lambda x : C_1.\ e_1 \ \Rightarrow \ \Psi_2 \to D_2 \tag{57}$$

Namely,

$$\Gamma_2 \mid \Psi_2', D_1 \vdash \lambda x : C_1.\ e_1 \ \Rightarrow \ D_1 \to \Psi_2' \to D_2 \tag{58}$$

This holds by applying the typing rule T-LAMANN2, with equation 51, and use subtyping transitivity (lemma 26) on equation 47, 40 to get $D_1 \ <: \ C_1$. The rest part of the conclusion is already proved (equation 56). So we are done.

– Case LAM. Similar to LAMANN2. According to the typing rule and hypothesis, we have, for some C_1 and Ψ_1',

$$e = \lambda x.\ e_1 \tag{59}$$
$$\Psi_1 = \Psi_1', C_1 \tag{60}$$
$$\Gamma_1, x : C_1 \mid \Psi_1' \vdash e_1 \ \Rightarrow \ \Psi_1' \to A_1 \tag{61}$$
$$A_2 = \Gamma_{1gen}(A_1) = \forall \bar{a}.A_1 \tag{62}$$
$$\Psi_2 \ <: \ \Psi_1', C_1 \tag{63}$$
$$ftv(\Psi_1', C_1) \subseteq ftv(\Gamma_1) \tag{64}$$

where $\bar{a} = ftv(A_1) - ftv(\Gamma_1)$. Because from equation 64 we know $ftv(C_1) \subseteq ftv(\Gamma_1)$, we get $\bar{a} = ftv(A_1) - ftv(\Gamma_1, C_1)$. So

$$\overline{(\Gamma_1, x : C_1)(A_1)} = \forall \bar{a}.A_1. \tag{65}$$

Do inversion on equation 63, we get for some D_1 and Ψ_2',

$$\Psi_2 = \Psi_2', D_1 \tag{66}$$
$$D_1 \ <: \ C_1 \tag{67}$$
$$\Psi_2' \ <: \ \Psi_1' \tag{68}$$

From equation 64, it is easy to derive that

$$ftv(\Psi_1') \subseteq ftv(\Gamma_1, x : C_1) \tag{69}$$

We have $ftv(\Psi_2) \subseteq ftv(\Gamma_2)$, combining equation 66, we can derive

$$ftv(\Psi_2') \subseteq ftv(\Gamma_2, x : D_1) \tag{70}$$

From $\Psi_2 \ <: \ \Psi_1$ and equation 67,

$$\Gamma_2, x : D_1 \ <: \ \Gamma_1, x : C_1 \tag{71}$$

Applying hypothesis on equation 61, equation 71, equation 65, equation 68, equation 69, equation 70, we get for some D_2,

$$\Gamma_2, x : D_1 \mid \Psi_2' \vdash e_1 \ \Rightarrow \ \Psi_2' \to D_2 \tag{72}$$

$$(\Gamma_2, x : D_1)_{gen}(D_2) = \forall \bar{b}.D_2 \tag{73}$$

$$\forall \bar{b}.D_2 \ <: \ \forall \bar{a}.A_1 \tag{74}$$

where $\bar{b} = ftv(D_2) - ftv(\Gamma_2, x : D_1)$. From equation 66 and $ftv(\Psi_2) \subseteq ftv(\Gamma_2)$, we know $ftv(D_1) \subseteq ftv(\Gamma_2)$. So $\bar{b} = ftv(D_2) - ftv(\Gamma_2)$. So according to lemma 32, we have

$$\Gamma_{2gen}(D_2) = \forall \bar{b}.D_2 \tag{75}$$

What we want is

$$\Gamma_{2gen}(D_2) \ <: \ \Gamma_{1gen}(A_1) \tag{76}$$

Namely,

$$\forall \bar{b}.D_2 \ <: \ \forall \bar{a}.A_1 \tag{77}$$

This is exactly equation 74.
Finally, feed D_2 and $\forall \bar{b}.D_2$ to the conclusion, we could get what we want is

$$\Gamma_2 \mid \Psi_2 \vdash \lambda x. \ e_1 \ \Rightarrow \ \Psi_2 \to D_2 \tag{78}$$

Namely,

$$\Gamma_2 \mid \Psi_2', D_1 \vdash \lambda x. \ e_1 \ \Rightarrow \ D_1 \to \Psi_2' \to D_2 \tag{79}$$

This holds by applying the typing rule T-Lam, with equation 72. The rest part of the conclusion is already proved (equation 77). So we are done.

– Case LAM2. According to the typing rule and hypothesis, we have, for some τ and C_1,

$$e = \lambda x. \ e_1 \tag{80}$$

$$\Psi_1 = \Psi_2 = \emptyset \tag{81}$$

$$A_1 = \tau \to C_1 \tag{82}$$

$$\Gamma_1, x : \tau \vdash e_1 \ \Rightarrow \ C_1 \tag{83}$$

$$A_2 = \Gamma_{gen}(\tau \to C_1) = \forall \bar{a}.\tau \to C_1 \tag{84}$$

where $\bar{a} = ftv(\tau \to C_1) - ftv(\Gamma)$. We could split $\bar{a}$ into two part $\bar{a} = \bar{a}_1\bar{a}_2$, where $\bar{a}_1 = ftv(\tau) - ftv(\Gamma)$ represents the free type variables in τ, and

$\bar{a}_2 = ftv(C_1) - ftv(\Gamma) - ftv(\tau)$ represents the free type variables in C_1 and gets rid of the repeated free variables appearing in both C_1 and τ.
According to lemma 30, we have a L we can use for substitution.
Consider fresh variables $\bar{b}_1 \notin (L \cup ftv(\Gamma_1, \Gamma_2, \tau, C_1))$. Substitute $\bar{a}_1$ to $\bar{b}_1$ in equation 83, where we already know that $\bar{a}_1 \notin \Gamma_1$,

$$\Gamma_1, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \vdash e_1 \;\Rightarrow\; C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{85}$$

Because we have $\Gamma_2 \;<:\; \Gamma_1$, so,

$$\Gamma_2, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \;<:\; \Gamma_1, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{86}$$

Because $\bar{a}_2 = ftv(C_1) - ftv(\Gamma_1) - ftv(\tau)$, from the definition of generalization (lemma 32), we get

$$\Gamma_1, x : \tau_{gen}(C_1) = \forall \bar{a}_2 . C_1 \tag{87}$$

According to generalization substitution (lemma 31), we could substitute $\bar{a}_1$ to $\bar{b}_1$ in equation 87, because we already know $\bar{b}_1 \notin ftv(\Gamma_1, \tau, C_1)$. And Γ_1 stays the same because $\bar{a}_1 \notin ftv(\Gamma_1)$

$$(\Gamma, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!])_{gen}(C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!]) = \forall \bar{a}_2 . C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{88}$$

Applying hypothesis to equation 85, 86, 88, with empty application contexts, we get for some C_2

$$\Gamma_2, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \vdash e_1 \;\Rightarrow\; C_2 \tag{89}$$

$$\Gamma_2, x : \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]_{gen}(C_2) = \forall \bar{c} . C_2 \tag{90}$$

$$\forall \bar{c} . C_2 \;<:\; \forall \bar{a}_2 . C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{91}$$

where $\bar{c} = ftv(C_2) - ftv(\Gamma_2) - ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!])$.
Because τ is a monotype, it is easy to derive $\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]$ is a monotype. From equation 89, applying the lam rule, we could get

$$\Gamma_2 \vdash \lambda x . e_1 \;\Rightarrow\; \tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2 \tag{92}$$

Feed $(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2)$ and $\Gamma_{2gen}(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2)$ to the conclusion, then what we want to prove is

$$\Gamma_{2gen}(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2) \;<:\; \Gamma_{1gen}(\tau \to C_1) \tag{93}$$

with $\Gamma_{1gen}(\tau \to C_1) = \forall \bar{a} . \tau \to C_1 = \forall \bar{a}_1 \bar{a}_2 . \tau \to C_1$, namely,

$$\Gamma_{2gen}(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2) \;<:\; \forall \bar{a}_1 \bar{a}_2 . \tau \to C_1 \tag{94}$$

In order to get $\Gamma_{2gen}(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2)$, we need to know $ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2) - ftv(\Gamma_2)$, which can be split into $(ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]) - ftv(\Gamma_2)) + (ftv(C_2) - ftv(\Gamma_2) - ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!])) = (ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]) - ftv(\Gamma_2)) + \bar{c}$.

We know $\bar{a}_1 \in ftv(\tau)$, so $\bar{b}_1 \in ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!])$. From equation 90, $\bar{b}_1 \cap \bar{c} = \emptyset$. And we know $\bar{b}_1 \notin ftv(\Gamma_2)$. So $\bar{b}_1 \subseteq (ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]) - ftv(\Gamma_2))$. Consider $\bar{b}_2 = (ftv(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!]) - ftv(\Gamma_2)) - \bar{b}_1$, then we get

$$\Gamma_{2gen}(\tau[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \to C_2) = \forall \bar{b}_2 \bar{b}_1 \bar{c}.C_2 \tag{95}$$

Substitute equation 95 into equation 94, now what we want to prove is

$$\forall \bar{b}_2 \bar{b}_1 \bar{c}.C_2 \ <: \ \forall \bar{a}_1 \bar{a}_2.C_1 \tag{96}$$

Surely,

$$\forall \bar{b}_2 \bar{b}_1 \bar{c}.C_2 \ <: \ \forall \bar{b}_1 \bar{c}.C_2 \tag{97}$$

by repeatedly apply S-ForallL. So we can turn to prove

$$\forall \bar{b}_1 \bar{c}.C_2 \ <: \ \forall \bar{a}_1 \bar{a}_2.C_1 \tag{98}$$

and use subtyping transitivity (lemma 26) with equation 97, 98 to finish the proof of equation 96.
In order to prove equation 98, notice we could α renaming $\bar{a}_1$ to $\bar{b}_1$, then it becomes

$$\forall \bar{b}_1 \bar{c}.C_2 \ <: \ \forall \bar{b}_1 \bar{a}_2.C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{99}$$

By repeatedly use S-ForallR and S-ForallL, we could remove the $\bar{b}_1$ and get

$$\forall \bar{c}.C_2 \ <: \ \forall \bar{a}_2.C_1[\![\bar{a}_1 \mapsto \bar{b}_1]\!] \tag{100}$$

This is exactly equation 91 so we are done.

– Case APP. According to the typing rule and hypothesis, we have, for some e_1, e_2 and C_1, C_2,

$$e = e_1 \ e_2 \tag{101}$$
$$\Gamma_1 \vdash e_1 \ \Rightarrow \ C_1 \tag{102}$$
$$\Gamma_{1gen}(C_1) = C_2 \tag{103}$$
$$\Gamma_1 \shortmid \Psi_1, C_2 \vdash e_1 \ \Rightarrow \ C_2 \to \Psi_1 \to A_1 \tag{104}$$
$$A_2 = \Gamma_{1gen}(A_1) = \forall \bar{a}.A_1 \tag{105}$$
$$\Psi_2 \ <: \ \Psi_1 \tag{106}$$

where $\bar{a} = ftv(A_1) - ftv(\Gamma_1)$.
Applying hypothesis on equation 102 with empty application contexts, and , $\Gamma_2 \ <: \ \Gamma_1$, we gets for some D_1 and D_2,

$$\Gamma_2 \vdash e_1 \ \Rightarrow \ D_1 \tag{107}$$
$$\Gamma_{2gen}(D_1) = D_2 \tag{108}$$
$$D_2 \ <: \ C_2 \tag{109}$$

From $\Psi_2 <: \Psi_1$ and $D_2 <: C_2$, we get

$$\Psi_2, D_2 <: \Psi_1, C_2 \tag{110}$$

Equation 104 can also be regarded as

$$\Gamma_1 \shortmid \Psi_1, C_2 \vdash e_1 \Rightarrow (\Psi_1, C_2) \to A_1 \tag{111}$$

From equation 103, because C_2 is generalized type under Γ_1, so $ftv(C_2) \subseteq ftv(\Gamma_1)$. We know $ftv(\Psi_1) \subseteq ftv(\Gamma_1)$, so $ftv(\Psi_1, C_2) \subseteq ftv(\Gamma_1)$. Similarly, $ftv(\Psi_2, D_2) \subseteq ftv(\Gamma_2)$.
Apply hypothesis on equation 111, with $\Gamma_2 <: \Gamma_1$, equation 110, we get for some D_3 and D_4

$$\Gamma_2 \shortmid \Psi_2, D_2 \vdash e_1 \Rightarrow (\Psi_2, D_2) \to D_3 \tag{112}$$
$$D_4 = \overline{\Gamma_2(D_3)} \tag{113}$$
$$D_4 <: \forall \bar{a}.A_1 \tag{114}$$

Feed D_3 and D_4 to the conclusion, we get what we want is

$$\Gamma_2 \shortmid \Psi_2 \vdash e_1\ e_2 \Rightarrow \Psi_2 \to D_3 \tag{115}$$
$$D_4 = \Gamma_{2gen}(D_3) \tag{116}$$
$$D_4 <: \forall \bar{a}.A_1 \tag{117}$$

Equation 116 and 117 are exactly equation 113 and 114. Equation 115 holds because we can apply typing rule for application with equation 107, 108 and 112. So we are done.

□

D.4 Extension of Pairs

We have three more cases for the typing derivation.

– Case Pair. According to the typing rule of pair, we have

$$e = (e_1, e_2) \tag{118}$$
$$\Psi_1 = \varnothing \tag{119}$$
$$A_1 = (C_1, C_2) \tag{120}$$
$$A_2 = \forall \bar{a}.A_1 \tag{121}$$

Where $\bar{a} = ftv(A_1) - ftv(\Gamma_1)$. By Lemma 32, we have for some C_2, C_3,

$$C_3 = \overline{\Gamma_1(C_1)} = \forall \overline{a_1}.C_1 \tag{122}$$
$$C_4 = \overline{\Gamma_1(C_2)} = \forall \overline{a_2}.C_2 \tag{123}$$

Where $\overline{a_1} = ftv(C_1) - ftv(\Gamma_1)$, and $\overline{a_2} = ftv(C_1) - ftv(\Gamma_1)$, and $\bar{a}_1 \cup \bar{a}_2 = \bar{a}$.

By hypothesis, we have for some D_1, D_2,

$$\Gamma_2 \vdash e_1 \Rightarrow D_1 \tag{124}$$
$$\overline{\Gamma_2(D_1)} = D_3 = \forall \overline{b_1}.D_1 \tag{125}$$
$$\Gamma_2 \vdash e_2 \Rightarrow D_2 \tag{126}$$
$$\overline{\Gamma_2(D_2)} = D_4 = \forall \overline{b_2}.D_2 \tag{127}$$
$$\Gamma_2 \vdash D_3 <: C_3 \tag{128}$$
$$\Gamma_2 \vdash D_4 <: C_4 \tag{129}$$

Where $\overline{b_1} = ftv(D_1) - ftv(\Gamma_2)$, and $\overline{b_2} = ftv(D_1) - ftv(\Gamma_2)$.
So we have $\overline{b_1} \notin ftv(\Gamma_2)$. By Lemma 30, there exists L. Choose fresh $\bar{c} \notin L$, we have

$$\Gamma_2 \vdash e_1 \Rightarrow D_1[\![\overline{b_1} \mapsto \bar{c}]\!] \tag{130}$$
$$\overline{\Gamma_2(D_1[\![\overline{b_1} \mapsto \bar{c}]\!])} = \forall \bar{c}.D_1 \tag{131}$$

So by typing rules for pair, combining with equation 130 and 126, we have

$$\Gamma_2 \vdash (e_1, e_2) \Rightarrow (D_1[\![\overline{b_1} \mapsto \bar{c}]\!], D_2) \tag{132}$$
$$\overline{\Gamma_2((D_1[\![\overline{b_1} \mapsto \bar{c}]\!], D_2))} = \forall \bar{c}\overline{b_2}.(D_1, D_2) \tag{133}$$

By alpha-equality on equation 128, we have

$$\Gamma_2 \vdash \forall \bar{c}.D_1 <: C_3 \tag{134}$$

By equation 129 and equation 134, it is easy to verify that

$$\Gamma_2 \vdash \forall \bar{c}\overline{b_2}.(D_1, D_2) <: \forall \bar{a}.(C_1, C_2) \tag{135}$$

What we want is equation 132 and 135 so we are done.

– Case Fst. According to the typing rule of **fst**, we have

$$e = \mathbf{fst} \tag{136}$$
$$\Psi_1 \vdash \forall ab.(a, b) \to a <: \Psi_1 \to A_1 \tag{137}$$
$$A_2 = \forall \bar{c}.A_1 \tag{138}$$

Where $\bar{c} = ftv(A_1) - ftv(\Gamma_1)$. Because we have $ftv(\Psi_1) \subseteq ftv(\Gamma_1)$, so $\bar{c} \notin \Psi_1$.
By subtyping substitution (Lemma 27), for some fresh $\bar{d}$, we have

$$\Psi_1 \vdash \forall ab.(a, b) \to a <: \Psi_1 \to A_1[\![\bar{c} \mapsto \bar{d}]\!] \tag{139}$$

By Lemma 23, and $\Psi_2 <: \Psi_1$, we have for some C,

$$\Psi_2 \vdash \forall ab.(a, b) \to a <: \Psi_2 \to C \tag{140}$$
$$C <: A_1[\![\bar{c} \mapsto \bar{d}]\!] \tag{141}$$

Because generalized result is more polymorphic than itself (lemma 28), so we have

$$\Gamma_{2gen}(C) \ <: \ C \tag{142}$$

According to subtyping transitivity (lemma 26), from equation141, 142, we get

$$\Gamma_{2gen}(C) \ <: \ A_1[\![\bar{c} \mapsto \bar{d}]\!] \tag{143}$$

Because from the principle of generalization, $ftv(\Gamma_{2gen}(C)) \subseteq ftv(\Gamma_2)$, and we have $\bar{d} \notin ftv(\Gamma_2)$, so

$$\bar{d} \notin ftv(\Gamma_{gen}(C)) \tag{144}$$

$$\Gamma_{2gen}(C) \ <: \ \forall \bar{d}.A_1[\![\bar{c} \mapsto \bar{d}]\!] \tag{145}$$

Equation 145 holds because $\bar{d}$ are themselves fresh to $\Gamma_{2gen}(C)$, so we could repeatedly apply rule S-ForallR and use equation 143 to get the subtyping relationship.
Then by α renaming of $\bar{d}$ to $\bar{c}$, we have

$$\Gamma_{2gen}(C) \ <: \ \forall \bar{c}.A_1 \tag{146}$$

So by the typing rule of **fst**, equation 140 and 146 we are done.

– Case snd. Similar as fst.

□

E Formalized Type Inference Algorithm

The specification given in the paper does not directly lead to an algorithm. So here we give a formalized algorithm corresponding to our declarative system. And we also prove it is sound and complete with respect to the declarative system.

Our algorithm is based on the work by [27, 35], which is extended to deal with application contexts in typing and subtyping. We have many components that are similar to theirs. Specifically, the unification algorithm, the arrow unification algorithm, and half of the subtyping algorithm are almost the same as theirs. Since those components are very standard we only give a brief overview of our formalization, while highlighting some of the differences. Actually the proofs involving application contexts bring additional complexities.

E.1 Notation

In the declarative system, we are enjoying guessing too much. For example, in rule S-ForallL, we are instantiating the polymorphic type with a guessed monotype τ. However, in an algorithm, we need to specify what we are going to provide. Therefore, we use the notation meta type variable $\widehat{\alpha}, \widehat{\beta}$, which stands for a unknown type that needs to be solved later. Now in our S-ForallL, we instantiate the polymorphic type with the meta type variable $\widehat{\alpha}$. Later on, we may meet some constraints on $\widehat{\alpha}$ which help us solve it. Therefore, each time we generate a new meta variable, we need to guarantee it is fresh. Also, each time we make a type variable when we open a polymorphic type, we need to make sure it is fresh.

So, we will use a source for name generating during all the processes, which is represented as N. We use the same name generator for both meta type variable and typing variable for clarity. In each process, there will be a name generator given, and the process uses as many fresh names as it needs, then returns back the name generator. It will make clearer when we see the concrete rules.

Also, we need substitutions to record the solution for meta type variables, represent as S. The substitution works as a finite map, which maps meta type variable to its solution. The substitution will keep growing as meta type variables are solved. The solution can be changed if the solution involves some other meta type variables that are newly solved, then the newly solved meta type variables in this solution will be substituted by their respective solutions. Again, each process will take current substitution as input, and pass it around while calling other process, than return the grown one back.

We use fov to mean the free ordinary variables and ftv to mean free type variables and ordinary variables.

With those notations, now we can dive into the algorithm.

E.2 Algorithm

$\boxed{S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1}$
τ_1, τ_2 inputs

$$\frac{\tau = \widehat{\alpha}\ or\ \tau = a\ or\ \tau = \mathsf{Int}}{S_0 \vdash \tau = \tau \hookrightarrow S_0}\ \text{AU-Refl}$$

$$\frac{\widehat{\alpha} \in dom(S_0) \qquad S_0 \vdash S_0\widehat{\alpha} = \tau \hookrightarrow S_1}{S_0 \vdash \widehat{\alpha} = \tau \hookrightarrow S_1}\ \text{AU-BVar1} \qquad \frac{\widehat{\alpha} \in dom(S_0) \qquad S_0 \vdash S_0\widehat{\alpha} = \tau \hookrightarrow S_1}{S_0 \vdash \tau = \widehat{\alpha} \hookrightarrow S_1}\ \text{AU-BVar2}$$

$$\frac{\widehat{\alpha} \notin dom(S_0) \qquad \widehat{\alpha} \notin ftv(S_0\tau)}{S_0 \vdash \widehat{\alpha} = \tau \hookrightarrow [\![\widehat{\alpha} \mapsto S_0\tau]\!] \cdot S_0}\ \text{AU-Var1} \qquad \frac{\widehat{\alpha} \notin dom(S_0) \qquad \widehat{\alpha} \notin ftv(S_0\tau)}{S_0 \vdash \tau = \widehat{\alpha} \hookrightarrow [\![\widehat{\alpha} \mapsto S_0\tau]\!] \cdot S_0}\ \text{AU-Var2}$$

$$\frac{S_0 \vdash \tau_1 = \tau_1' \hookrightarrow S_1 \qquad S_1 \vdash \tau_2 = \tau_2' \hookrightarrow S_2}{S_0 \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' \hookrightarrow S_2}\ \text{AU-Fun}$$

Fig. 13. Unification

Unification Unification process is given in Figure 13. The form $S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1$ means, given current substitution S_0, try to unify τ_1 and τ_2, which has the result S_1. This process is very standard, as it is in Hindley-Milner algorithm. Rule AU-Refl is when two sides are the same, so we return the input substitution back. Both in rule AU-BVar1 and AU-Var1, the left hand side is a meta type variable $\widehat{\alpha}$. But in AU-BVar1, $\widehat{\alpha}$ is already solved in the input substitution, so all we need to do is to unify its solution with the right hand side. In AU-Var1, we don't have the solution for $\widehat{\alpha}$ yet, which means now we can solve $\widehat{\alpha}$ with right hand side, under the condition that $\widehat{\alpha} \notin ftv(S_0\tau)$, which is to ensure there is no recursive solution such as $\widehat{\alpha} = \widehat{\alpha} \to \mathsf{Int}$. Notice the notation for substitution composition $\cdot$ means we extend the substitution with the new item $\widehat{\alpha}$ with its solution $S_0\tau$, meanwhile, we will substitution all $\widehat{\alpha}$ appearing in other solutions with $S_0\tau$. Rule AU-BVar and AU-Var2 works similarly. In rule AU-Fun, we will unify the argument type first, and use the return substitution as input substitution when we unify the return type.

$\boxed{(S_0, N_0) \vdash^{\to} A = A_1 \to A_2 \hookrightarrow (S_1, N_1)}$
A input, $A_1 \to A_2$ outputs

$$\frac{S_0 \vdash \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1}{(S_0, N_0\widehat{\alpha}_1\widehat{\alpha}_2) \vdash^{\to} \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow (S_1, N_0)}\ \text{AF-Mono}$$

$$\frac{}{(S_0, N_0) \vdash^{\to} A \to B = A \to B \hookrightarrow (S_0, N_0)}\ \text{AF-Arrow}$$

Fig. 14. Arrow Unification

Arrow Unification There are cases we want a certain type be the form of a function type, for example, if in the subtyping relation, the left hand side is a function type, we expect the right hand side to be also a function type.

Therefore, we have another form of unification, which is arrow unification, where given a input type, we will return a function type back, as shown in Figure 14. Rule AF-Mono is when the type is a meta type variable, then we will call unification to split the type into a function type and return the function type back. Rule AF-Arrow is when the type is already a function type, then we only return it back.

We will see its usage later.

$\boxed{(S_0, N_0) \vdash A \ <: \ B \hookrightarrow (S_1, N_1)}$
A, B inputs

$$\frac{(S_0, N_0) \vdash A \ <: \ B[\![a \mapsto b]\!] \hookrightarrow (S_1, N_1) \qquad b \notin ftv(S_1 A) \qquad b \notin ftv(S_1(\forall a.B))}{(S_0, N_0 b) \vdash A \ <: \ \forall a.B \hookrightarrow (S_1, N_1)} \text{AS-ForallR}$$

$$\frac{(S_0, N_0) \vdash A[\![a \mapsto \widehat{\beta}]\!] \ <: \ B \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}) \vdash \forall a.A \ <: \ B \hookrightarrow (S_1, N_1)} \text{AS-ForallL}$$

$$\frac{\begin{array}{c}(S_0, N_0) \vdash^{\rightarrow} A = A_1 \to A_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash B \ <: \ A_1 \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash A_2 \ <: \ C \hookrightarrow (S_3, N_3)\end{array}}{(S_0, N_0) \vdash A \ <: \ B \to C \hookrightarrow (S_3, N_3)} \text{AS-FunR}$$

$$\frac{\begin{array}{c}(S_0, N_0) \vdash^{\rightarrow} A = A_1 \to A_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash A_1 \ <: \ B \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash C \ <: \ A_2 \hookrightarrow (S_3, N_3)\end{array}}{(S_0, N_0) \vdash B \to C \ <: \ A \hookrightarrow (S_3, N_3)} \text{AS-FunL}$$

$$\frac{S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1}{(S_0, N_0) \vdash \tau_1 \ <: \ \tau_2 \hookrightarrow (S_1, N_0)} \text{AS-Mono}$$

$\boxed{(S_0, N_0) \mid \Psi \vdash A \ <: \ B \hookrightarrow (S_1, N_1)}$
Ψ, A input, B output

$$\frac{}{(S_0, N_0) \mid \emptyset \vdash A \ <: \ A \hookrightarrow (S_0, N_0)} \text{AS-Empty} \qquad \frac{(S_0, N_0) \mid \Psi \vdash A[\![a \mapsto \widehat{\beta}]\!] \ <: \ B \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}) \mid \Psi \vdash \forall a.A \ <: \ B \hookrightarrow (S_1, N_1)} \text{AS-ForallL2}$$

$$\frac{(S_0, N_0) \vdash C \ <: \ A \hookrightarrow (S_1, N_1) \qquad (S_1, N_1) \mid \Psi \vdash B \ <: \ D \hookrightarrow (S_2, N_2)}{(S_0, N_0) \mid \Psi, C \vdash A \to B \ <: \ C \to D \hookrightarrow (S_2, N_2)} \text{AS-Fun2}$$

$$\frac{(S_0, N_0) \vdash^{\rightarrow} \tau = \tau_1 \to \tau_2 \hookrightarrow (S_1, N_1) \qquad (S_1, N_1) \mid \Psi \vdash \tau_1 \to \tau_2 \ <: \ B \hookrightarrow (S_2, N_2)}{(S_0, N_0) \mid \Psi \vdash \tau \ <: \ B \hookrightarrow (S_2, N_2)} \text{AS-Mono2}$$

Fig. 15. Subtyping

Subtyping Subtyping algorithm corresponds to the declarative subtyping shown in Figure 15, the main difference is we stop guessing. Rule AS-ForallR uses a fresh name from name generator as the type variable in the right hand side. But since b is generated only in the scope of the polymorphic type, if any of the types after substituted by S_1 contains b, it means the left hand side is actually not more polymorphic than the right hand side and the unification will fail. In rule AS-ForallL, we use a fresh name but this time as a meta type variable waiting to be solved. Comparing to the declarative system, where we guess a τ, we use the meta type variable to represent the unknown type and continue the algorithm. Rule AS-FunR and AS-FunL works similarly, which depend on arrow unification. Rule AS-Mono deals with all the rest cases, where these two types must be unified.

The second part of the subtyping works similarly. And most cases correspond to the one in declarative system. Rule AS-Mono deals with the case when the left hand side is a meta type variable. In this case, we need to use arrow unification to split the meta type variable into the function form and continue the subtyping.

Typing Figure 16 shows the algorithm for typing, along with the generalization process. Most cases are straightforward. Rule AT-Var relies on subtyping. AT-Int is trivial. Rule AT-Lamann1 and AT-Lamann2 follows directly from declarative system. In rule AT-Lam1, again, we use a fresh meta type variable to represent the unknown type. Rule AT-Lam2 and AT-App are also easy-to-follow cases.

E.3 Properties and proofs

In this section, we state the import lemmas and theories about the algorithm system, and give the proofs. First, we need some auxiliary definitions and notations to help with the proofs.

Auxiliary

Definition 2 (Well-defined of Substitution). *S is well-defined iff $\forall A. SA = S(SA)$, and there is no item $[\![\widehat{\alpha} \mapsto \widehat{\alpha}]\!]$ in S.*

For example, $S_0 = (\widehat{\alpha} = \mathsf{Int}, \widehat{\beta} = \mathsf{Int})$ is well-defined, meanwhile $S_1 = (\widehat{\alpha} = \mathsf{Int}, \widehat{\beta} = \widehat{\alpha})$ is not, since $S_1\widehat{\beta} = \widehat{\alpha}$, and $S_1 S_1 \widehat{\beta} = \mathsf{Int}$. Also, according to the definition, surely for any well-defined S, $dom(S)$ is disjoint with $range(S)$.

Through the proofs, we will assume all substitutions are well-defined. Since we require all the input substitutions are well-defined, and also it is not complicated to show that in all the relationships, if the input substitution is well-defined, then the output substitution is also well-defined. So we omit related proofs.

Similarly, we will assume all variables in name supply N are fresh enough to all existing variables.

$\boxed{(S_0, N_0) \vdash \Gamma_{agen}(A) = B \hookrightarrow (S_0, N_1)}$
Γ, A inputs, B output

$$\frac{\bar{\widehat{\alpha}} = ftv(S_0 A) - ftv(S_0 \Gamma)}{(S_0, N_0 \bar{b}) \vdash \Gamma_{agen}(A) = \forall \bar{b}.(S_0 A)[\![\bar{\widehat{\alpha}} \mapsto \bar{b}]\!] \hookrightarrow (S_0, N_0)} \text{ AT-Gen}$$

$\boxed{(S_0, N_0) \mid \Gamma \mid \Psi \vdash e \;\Rightarrow\; A \hookrightarrow (S_1, N_1)}$
Γ, Ψ, e inputs, A output

$$\frac{x : A \in \Gamma \qquad (S_0, N_0) \mid \Psi \vdash A \;<:\; B \hookrightarrow (S_1, N_1)}{(S_0, N_0) \mid \Gamma \mid \Psi \vdash x \;\Rightarrow\; B \hookrightarrow (S_1, N_1)} \text{ AT-Var}$$

$$\frac{}{(S_0, N_0) \mid \Gamma \vdash n \;\Rightarrow\; \mathsf{Int} \hookrightarrow (S_0, N_0)} \text{ AT-Int}$$

$$\frac{(S_0, N_0) \mid \Gamma, x : A \vdash e \;\Rightarrow\; B \hookrightarrow (S_1, N_1)}{(S_0, N_0) \mid \Gamma \vdash \lambda x : A.\ e \;\Rightarrow\; A \to B \hookrightarrow (S_1, N_1)} \text{ AT-LamAnn1}$$

$$\frac{(S_0, N_0) \vdash C \;<:\; A \hookrightarrow (S_1, N_1) \qquad (S_1, N_1) \mid \Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B \hookrightarrow (S_2, N_2)}{(S_0, N_0) \mid \Gamma \mid \Psi, C \vdash \lambda x : A.\ e \;\Rightarrow\; C \to B \hookrightarrow (S_2, N_2)} \text{ AT-LamAnn2}$$

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

$$\frac{(S_0, N_0) \mid \Gamma, x : A \mid \Psi \vdash e \;\Rightarrow\; B \hookrightarrow (S_1, N_1)}{(S_0, N_0) \mid \Gamma \mid \Psi, A \vdash \lambda x.\ e \;\Rightarrow\; A \to B \hookrightarrow (S_1, N_1)} \text{ AT-Lam2}$$

$$\frac{\begin{array}{c}(S_0, N_0) \mid \Gamma \vdash e_2 \;\Rightarrow\; A \hookrightarrow (S_1, N_1) \qquad (S_1, N_1) \vdash \Gamma_{agen}(A) = B \hookrightarrow (S_2, N_2) \\ (S_2, N_2) \mid \Gamma \mid \Psi, B \vdash e_1 \;\Rightarrow\; B \to C \hookrightarrow (S_3, N_3)\end{array}}{(S_0, N_0) \mid \Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; C \hookrightarrow (S_3, N_3)} \text{ AT-App}$$

Fig. 16. Typing

Since substitutions are maps, we regard the permutation of domain produces the same substitution, namely $(\widehat{\alpha} = \mathsf{Int}, \widehat{\beta} = \mathsf{Int}) = (\widehat{\beta} = \mathsf{Int}, \widehat{\alpha} = \mathsf{Int})$. And we have a weaker equivalence definition:

Definition 3 (Excluded-χ Equivalence of Substitution). *$S_1 = S_2 \backslash_\chi$ means substitutions S_1 and S_2 are equal except for variable set χ. Namely, $\forall \widehat{\alpha} \notin \chi$, then $S_1 \widehat{\alpha} = S_2 \widehat{\alpha}$.*

During the proofs, we need the substitution lemmas.

Lemma 34 (Subtyping Substitution).

1. *If $A \;<:\; B$, then $S_1 A \;<:\; S_1 B$.*
2. *If $\Psi \vdash A \;<:\; B$, then $S_1 \Psi \vdash S_1 A \;<:\; S_1 B$.*

Proof. This lemma has two parts, and only the second part relies on the first one. So we can prove them separately.

Part 1 By induction on the subtyping relationship and do case analysis.

- Case S-INT. Holds trivially.
- Case S-VAR. Holds trivially if both sides are just ordinary variables.
- Case S-FORALLR. By induction hypothesis, we have $S_1 A \ <: \ S_1 B$. And $S_1 A \ <: \ S_1 \forall a.B$ can be rewritten as $S_1 A \ <: \ \forall a.S_1 B$ as we assume a is fresh enough, otherwise we can use α renaming to choose a fresh variable. Then by rule S-ForallR the case is finished.
- Case S-FUN. By induction hypothesis, we have $S_1 C \ <: \ S_1 A$ and $S_1 B \ <: \ S_1 D$. Follows directly by using rule S-Fun.
- Case S-FORALLL. By induction hypothesis, we have $S_1(A[\![a \mapsto \tau]\!]) \ <: \ S_1 B$. Namely, $(S_1 A)[\![a \mapsto (S_1 \tau)]\!] \ <: \ S_1 B$, since S_1 only contains meta type variables in domain. So we can derive $S_1(\forall a.A) \ <: \ S_1 B$, or $\forall a.S_1 A \ <: \ S_1 B$ by using rule S-ForallL with type $S_1 \tau$.

Part 2 By induction on the subtyping relationship and do case analysis.

- Case S-EMPTY. Holds trivially by applying S-Empty.
- Case S-FORALLL2. Similar as the one in part 1. By induction hypothesis, we have $S_1 \Psi \vdash S_1(A[\![a \mapsto \tau]\!]) \ <: \ S_1 B$. Namely, $S_1 \Psi \vdash (S_1 A)[\![a \mapsto (S_1 \tau)]\!] \ <: \ S_1 B$, since S_1 only contains meta type variables in domain. So we can derive $S_1 \Psi \vdash S_1(\forall a.A) \ <: \ S_1 B$, or $S_1 \Psi \vdash \forall a.S_1 A \ <: \ S_1 B$ by using rule S-ForallL2 with type $S_1 \tau$.
- Case S-FUN2. By induction hypothesis and the lemma of first part, we have $S_1 C \ <: \ S_1 A$, and $S_1 \Psi \vdash S_1 B \ <: \ S_1 D$. So $S_1(\Psi, C) \vdash S_1(A \to B) \ <: \ S_1(C \to D)$, namely $S\Psi, S_1 C \vdash S_1 A \to S_1 B \ <: \ S_1 C \to S_1 D$ holds by rule S-Fun2.

□

Lemma 35 (Typing Substitution).

1. *If* $\Gamma \shortmid \Psi \vdash e \ \Rightarrow \ t$, *then* $S_1\Gamma \shortmid S_1\Psi \vdash e \ \Rightarrow \ S_1 t$.
2. *If* $\Gamma \vdash e \ \Rightarrow \ t$, *and* $\Gamma_{gen}(t) = t_1$, *then* $\exists t_2$, *that* $S_1\Gamma \vdash e \ \Rightarrow \ t_2$, *and* $S_1\Gamma_{gen}(t_2) = S_1 t_1$.

Proof. This lemmas has two parts and they depends on each other. So we prove them simultaneously.

Part 1 By induction on the height of derivation, and case analyze the last rule used in the derivation.

- Case T-VAR. Since we have $x : A \in \Gamma$, so $x : S_1 A \in S_1\Gamma$. And by subtyping substitution $S_1 \Psi \vdash S_1 A \ <: \ S_1 B$. So using T-Var finishes the case.
- Case T-INT. Holds trivially.
- Case T-LAMANN. By induction hypothesis, we have $S_1(\Gamma, x : A) \vdash e \ \Rightarrow \ S_1 B$. Since A is a well defined type and contains no free variables, rewrite the equation we get $S_1\Gamma, x : A \vdash e \ \Rightarrow \ S_1 B$. So by using T-LAMANN, $S_1\Gamma \vdash \lambda x : A.\ e \ \Rightarrow \ A \to S_1 B$ holds.

- Case T-LAMANN2. By induction hypothesis, we have $S_1(\Gamma, x : A) \mid S_1\Psi \vdash e \Rightarrow S_1 B$. Since A is a well defined type and contains no free variables, rewrite the equation we get $S_1\Gamma, x : A \mid S_1\Psi \vdash e \Rightarrow S_1 B$. Given $C <: A$, by subtyping substitution, we have $S_1 C <: S_1 A$. So the goal $S_1\Gamma \mid S_1(\Psi, C) \vdash \lambda x : A.\ e \Rightarrow S_1(C \to B)$, namely $S_1\Gamma \mid S_1\Psi, S_1 C \vdash \lambda x : A.\ e \Rightarrow S_1 C \to S_1 B$ holds by rule T-LAMANN2.
- Case T-LAM. By induction hypothesis, we have $S_1(\Gamma, x : A) \mid S_1\Psi \vdash e \Rightarrow S_1 B$, namely $S_1\Gamma, x : S_1 A \mid S_1\Psi \vdash e \Rightarrow S_1 B$. So by using T-LAM, $S_1\Gamma \mid S_1(\Psi, A) \vdash \lambda x.\ e \Rightarrow S_1 A \to S_1 B$ holds.
- Case T-LAM2. By induction hypothesis, we have $S_1(\Gamma, x : \tau) \vdash e \Rightarrow S_1 B$, namely $S_1\Gamma, x : S_1\tau \vdash e \Rightarrow S_1 B$. So the goal $S_1\Gamma \vdash \lambda x.\ e \Rightarrow S_1(\tau \to B)$ holds by rule T-LAM2 with type $S_1\tau$.
- Case T-App. By preconditions and induction hypothesis, we have

$$\Gamma \vdash e_1 \Rightarrow A \tag{147}$$
$$\Gamma_{gen}(A) = B \tag{148}$$
$$S_1\Gamma \mid S_1\Psi, S_1 B \vdash e_1 \Rightarrow S_1 B \to S_1 C \tag{149}$$

  Here we use part 2 of this lemma on equation 147 and 148, get for some t,

$$S_1\Gamma \vdash e_1 \Rightarrow t \tag{150}$$
$$S_1\Gamma_{gen}(t) = S_1 B \tag{151}$$

  Then, by using typing rule for application on equation 150, 151 and 149, we get $S_1\Gamma \mid S_1\Psi \vdash e_1\ e_2 \Rightarrow S_1 C$ and we are done.
  Notice here we are using part 2 of the lemma with a smaller height, since height of equation 147 is smaller than original one (the application $e_1\ e_2$).

Part 2 Suppose $\bar{a} = ftv(t) - ftv(\Gamma)$, then $t_1 = \forall \bar{a}.t$. For some $\bar{b}$ which are fresh enough (with Γ, S_1, and t), applying part 1 of the lemma with $S_1 \cdot [\![\bar{a} \mapsto \bar{b}]\!]$, we get (we know $\bar{a} \notin ftv(\Gamma)$)

$$S_1\Gamma \vdash e \Rightarrow S_1 \cdot [\![\bar{a} \mapsto \bar{b}]\!]t \tag{152}$$

Namely,

$$S_1\Gamma \vdash e \Rightarrow S_1(t[\![\bar{a} \mapsto \bar{b}]\!]) \tag{153}$$

Now $\bar{b} = ftv(t[\![\bar{a} \mapsto \bar{b}]\!]) - ftv(\Gamma)$. And $\bar{b}$ is fresh to S_1, so $\bar{b} = ftv(S_1(t[\![\bar{a} \mapsto \bar{b}]\!])) - ftv(S_1\Gamma)$. Therefore, $(S_1\Gamma)_{gen}(S_1(t[\![\bar{a} \mapsto \bar{b}]\!])) = \forall \bar{b}.(S_1(t[\![\bar{a} \mapsto \bar{b}]\!])) = S_1(\forall \bar{b}.(t[\![\bar{a} \mapsto \bar{b}]\!]))$. By α renaming, $(\forall \bar{b}.(t[\![\bar{a} \mapsto \bar{b}]\!])) = (\forall \bar{a}.t)$. So choose $t_2 = S_1(t[\![\bar{a} \mapsto \bar{b}]\!])$, we get $(S_1\Gamma)_{gen}(t_2) = S_1(\forall \bar{a}.t) = S_1 t_1$.
Notice here we are applying the part 1 of this lemma with the same height. Namely, part 1 calls part 2 with smaller height, and part 2 calls part 1 with the same height. So this mutual proofs hold because the height is decreasing and it will terminate.

□

We use $\overline{\Gamma(A)} = \forall \bar{a}.A$, where $\bar{a} = ftv(A) - ftv(\Gamma)$, including both meta type variables and typing variables.

Lemma 36. $S\overline{\Gamma(A)} <: \overline{S\Gamma(SA)}$.

Proof. Assume $\overline{\Gamma(A)} = \forall \bar{a}.A$, where $\bar{a} = ftv(A) - ftv(\Gamma)$. Consider a fresh variable set $\bar{c}$, then by α renaming we have $\forall \bar{a}.A = \forall \bar{c}.A[\![\bar{a} \mapsto \bar{c}]\!]$. Since $\bar{c}$ are fresh so they not in S. So $S\overline{\Gamma(A)} = S(\forall \bar{c}.A[\![\bar{a} \mapsto \bar{c}]\!]) = \forall \bar{c}.S(A[\![\bar{a} \mapsto \bar{c}]\!])$.

Assume $\overline{S\Gamma(SA)} = \forall \bar{b}.SA$, where $\bar{b} = ftv(SA) - ftv(S\Gamma)$. Now our goal is $\forall \bar{c}.S(A[\![\bar{a} \mapsto \bar{c}]\!]) <: \forall \bar{b}.SA$.

We first prove $\bar{b} \notin ftv(\forall \bar{c}.S(A[\![\bar{a} \mapsto \bar{c}]\!]))$, or equivalently $\bar{b} \notin ftv(S\overline{\Gamma(A)})$, since then we can use $\bar{b}$ as fresh enough variables and turn to prove $\forall \bar{c}.S(A[\![\bar{a} \mapsto \bar{c}]\!]) <: SA$. The proof is done by contradiction. Assume exists a $b \in \bar{b}$, that $b \in ftv(S\overline{\Gamma(A)})$, then there must be a d, that $d \in ftv(\overline{\Gamma(A)})$, and $b \in ftv(Sd)$. Because $d \in ftv(\overline{\Gamma(A)})$, it must be the case $d \in ftv(\Gamma)$ and $d \in ftv(A)$. So already know $b \in ftv(Sd)$, and $d \in ftv(\Gamma)$, then $b \in ftv(S\Gamma)$. However, b comes from $ftv(SA) - ftv(S\Gamma)$. So, a contradiction.

Then we move to prove $\forall \bar{c}.S(A[\![\bar{a} \mapsto \bar{c}]\!]) <: SA$. By choose $\bar{\tau} = \overline{Sa}$, it is easy to deduce $(S(A[\![\bar{a} \mapsto \bar{c}]\!]))[\![\bar{c} \mapsto \overline{Sa}]\!] <: SA$ holds by reflexivity of subtyping. □

Soundness

Lemma 37 (Soundness of Unification). *if* $S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1$, *then* $S_1\tau_1 = S_2\tau_2$, *and* $\exists R$, *that* $S_1 = R \cdot S_0$. *And also* $vars(R) \subseteq ftv(S_0\tau_1, S_0\tau_2)$.

Proof. By induction on the derivation of unification.

- Case AU-REFL. Let $R = \emptyset$ and the goal follows trivially.
- Case AU-BVAR1. Given

$$S_0 \vdash S_0\widehat{\alpha} = \tau \hookrightarrow S_1 \tag{154}$$

  By induction hypothesis

$$S_1S_0\widehat{\alpha} = S_1\tau \tag{155}$$
$$S_1 = R \cdot S_0 \tag{156}$$
$$vars(R) \subseteq ftv(S_0S_0\widehat{\alpha}, S_0\tau) \tag{157}$$

  Because S_0 is well defined, so $S_1S_0\widehat{\alpha} = RS_0S_0\widehat{\alpha} = RS_0\widehat{\alpha} = S_1\widehat{\alpha}$. namely $S_1\widehat{\alpha} = S_1\tau$. Similarly, we could derive from equation 157 that $vars(R) \subseteq ftv(S_0\widehat{\alpha}, S_0\tau)$.

  Feed R to the goals and we already proved all the subgoals.
- Case AU-BVAR2. Similar to case AU-BVAR1.

– Case AU-VAR1. Pick $R = [\![\widehat{\alpha} \mapsto S_0\tau]\!]$, with

$$\widehat{\alpha} \notin dom(S_0) \tag{158}$$
$$\widehat{\alpha} \notin ftv(S_0\tau) \tag{159}$$

Because of equation 158, $S_0\widehat{\alpha} = \widehat{\alpha}$. So $[\![\widehat{\alpha} \mapsto S_0\tau]\!]S_0\widehat{\alpha} = [\![\widehat{\alpha} \mapsto S_0\tau]\!]\widehat{\alpha} = S_0\tau$, namely $S_1\widehat{\alpha} = S_0\tau$. Also because of equation 159, we get $[\![\widehat{\alpha} \mapsto S_0\tau]\!]S_0\tau = S_0\tau$, namely $S_1\tau = S_0\tau$. So combine this two derivation, we get $S_1\widehat{\alpha} = S_1\tau$. Moreover, $vars(R) = ftv(\widehat{\alpha}, S_0\tau) = ftv(S_0\widehat{\alpha}, S_0\tau)$.

– Case AU-VAR2. Similar to case AU-VAR2.

– Case AU-FUN. Given

$$S_0 \vdash \tau_1 = \tau_1' \hookrightarrow S_1 \tag{160}$$
$$S_1 \vdash \tau_2 = \tau_2' \hookrightarrow S_2 \tag{161}$$

Applying induction hypothesis, we could get $S_1\tau_1 = S_1\tau_1'$ and $S_1 = R_1S_0$ and $vars(R_1) \subseteq ftv(S_0\tau_1, S_0\tau_1')$. Also, $S_2\tau_2 = S_2\tau_2'$ and $S_2 = R_2S_1$ and $vars(R_2) \subseteq ftv(S_1\tau_2, S_1\tau_2')$. So $R_2S_1\tau_1 = R_2S_1\tau_1'$, namely $S_2\tau_1 = S_2\tau_1'$. Therefore it can be derived that $S_2(\tau_1 \to \tau_2) = S_2(\tau_1' \to \tau_2')$. And choose $R = R_2 \cdot R_1$.
Moreover, from $vars(R_2) \subseteq ftv(S_1\tau_2, S_1\tau_2')$, we can get $vars(R_2) \subseteq ftv(R_1S_0\tau_2, R_1S_0\tau_2')$. So $vars(R_2) \subseteq ftv(R_1) \cup ftv(S_0\tau_2, S_0\tau_2')$. And because $ftv(R_1) \subseteq ftv(S_0\tau_1, S_0\tau_1')$, we have $vars(R_2) \subseteq ftv(S_0\tau_1, S_0\tau_1') \cup ftv(S_0\tau_2, S_0\tau_2')$. This finishes the case.

□

Lemma 38 (Soundness of Arrow Unification). *if* $(S_0, N_0) \vdash^{\to} A = A_1 \to A_2 \hookrightarrow (S_1, N_1)$*, then* $S_1A = S_1A_1 \to S_1A_2$*, and* $\exists R$*, that* $S_1 = R \cdot S_0$*. Moreover,* $ftv(A_1 \to A_2)$*, and* $vars(S_1)$ *are all subsets of* $vars(S_0) \cup ftv(A) \cup (N_0 - N_1)$*;* $vars(R) \subseteq ftv(S_0A) \cup (N_0 - N_1)$*.*

Proof. By induction on arrow unification relation. We analyze each case.

– Case AF-MONO. $S_1\widehat{\alpha}_1 = S_1\widehat{\alpha}_1 \to S_1\widehat{\alpha}_2$ comes directly by the soundness of unification (lemma 37), with the same R. $ftv(\widehat{\alpha}_1 \to \widehat{\alpha}_2) = \{\widehat{\alpha}_1, \widehat{\alpha}_2\} = (N_0\widehat{\alpha}_1\widehat{\alpha}_2) - N_0$. So $ftv(\widehat{\alpha}_1 \to \widehat{\alpha}_2) \subseteq (N_0\widehat{\alpha}_1\widehat{\alpha}_2) - N_0$. And from soundness of unification, we get $vars(R) \subseteq ftv(S_0\widehat{\alpha}, S_0(\widehat{\alpha}_1 \to \widehat{\alpha}_2))$. Therefore $vars(R) \subseteq ftv(S_0\widehat{\alpha}) \cup ftv(\widehat{\alpha}_1 \to \widehat{\alpha}_2) \subseteq ftv(S_0\widehat{\alpha}) \cup ftv(N_0\widehat{\alpha}_1\widehat{\alpha}_2 - N_0)$.
Finally, from the soundness of unification, we know $S_1 = R \cdot S_0$. So $vars(S_1) \subseteq vars(R) \cup vars(S_0) \subseteq vars(S_0) \cup ftv(\widehat{\alpha}) \cup ftv(N_0\widehat{\alpha}_1\widehat{\alpha}_2 - N_0)$. The case is finished.

– Case AF-ARROW. Choose R to be empty, each goal holds trivially.

□

Lemma 39 (Soundness of Subtyping).

1. *if* $(S_0, N_0) \vdash A <: B \hookrightarrow (S_1, N_1)$*, then* $S_1A <: S_1B$*. And* $\exists R$*, that* $S_1 = R \cdot S_0$*. Moreover,* $vars(S_1) \subseteq vars(S_0) \cup ftv(A, B) \cup (N_0 - N_1)$*;* $vars(R) \subseteq ftv(S_0A, S_0B) \cup (N_0 - N_1)$*.*

2. *if* $(S_0, N_0) \vdash \Psi \vdash A \ <: \ B \hookrightarrow (S_1, N_1)$, *then* $S_1\Psi \vdash S_1 A \ <: \ S_1 B$. *And* $\exists R$, *that* $S_1 = R \cdot S_0$. *Moreover,* $vars(S_1)$ *and* $ftv(B)$ *are all subsets of* $vars(S_0) \cup ftv(\Psi) \cup ftv(A) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0 A) \cup ftv(S_0\Psi) \cup (N_0 - N_1)$.

Proof. This lemma has two parts, with only the second one depends on the first one. So we can prove them separately.

Part 1 We do induction on the subtyping relationship, and analyze each case.

- Case AS-FORALLR. By induction hypothesis we have

$$S_1 A \ <: \ S_1 B[\![a \mapsto b]\!] \tag{162}$$
$$S_1 = R \cdot S_0 \tag{163}$$

  where $vars(S_1) \subseteq vars(S_0) \cup ftv(A, B[\![a \mapsto b]\!]) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0 A, S_0(B[\![a \mapsto b]\!])) \cup (N_0 - N_1)$.
  Note $vars(S_0) \cup ftv(A, B[\![a \mapsto b]\!]) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(A, B) \cup (N_0 b - N_1)$. So both $vars(S_1) \subseteq vars(S_0) \cup ftv(A, B) \cup (N_0 b - N_1)$. Also, $ftv(S_0 A, S_0(B[\![a \mapsto b]\!])) \cup (N_0 - N_1) \subseteq ftv(S_0 A, S_0 B) \cup (N_0 b - N_1)$, so $vars(R) \subseteq ftv(S_0 A, S_0 B) \cup (N_0 b - N_1)$.
  Because substitution only contains meta variables, so $S_1 a = a$ and $S_1 b = b$. Therefore, $S_1 A \ <: \ S_1(\forall a.B)$ holds by renaming $\forall a.B$ to $\forall b.B[\![a \mapsto b]\!]$, and making substitution go inside forall since b is fresh enough, finally applying rule S-ForallR with equation 162.
- Case AS-FORALLL. By induction hypothesis we have

$$S_1(A[\![a \mapsto \widehat{\beta}]\!]) \ <: \ S_1 B \tag{164}$$
$$S_1 = R \cdot S_0 \tag{165}$$

  where $vars(S_1) \subseteq vars(S_0) \cup ftv(A, B[\![a \mapsto \widehat{\beta}]\!]) \cup ftv(N_0 - N_1)$, $vars(R) \subseteq ftv(S_0 A, S_0(B[\![a \mapsto \widehat{\beta}]\!])) \cup ftv(N_0 - N_1)$.
  We will assume $\bar{a}$ are fresh to S_1, then we can make substitution go inside one step in equation 164 to get

$$(S_1 A)[\![a \mapsto S_1\widehat{\beta}]\!]) \ <: \ S_1 B \tag{166}$$

  Our goal $S_1(\forall a.A) \ <: \ S_1 B$ holds by making substitution go inside one step to get $\forall a.S_1 A \ <: \ S_1 B$, and applying rule S-ForallL with $S_1\widehat{\beta}$. Moreover, $vars(S_0) \cup ftv(A, B[\![a \mapsto \widehat{\beta}]\!]) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(A, B) \cup \widehat{\beta} \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(A, B) \cup (N_0\widehat{\beta} - N_1)$. So we get both $vars(S_1) \subseteq vars(S_0) \cup ftv(A, B) \cup (N_0\widehat{\beta} - N_1)$.
  And $ftv(S_0 A, S_0(B[\![a \mapsto \widehat{\beta}]\!])) \cup ftv(N_0 - N_1) \subseteq ftv(S_0 A, S_0 B) \cup ftv(N_0\widehat{\beta} - N_1)$ since $\widehat{\beta}$ is fresh to S_0. So $vars(R) \subseteq ftv(S_0 A, S_0 B) \cup ftv(N_0\widehat{\beta} - N_1)$.
- Case AS-FUNR. By preconditions we have

$$(S_0, N_0) \vdash^{\rightarrow} A = A_1 \to A_2 \hookrightarrow (S_1, N_1) \tag{167}$$
$$(S_1, N_1) \vdash B \ <: \ A_1 \hookrightarrow (S_2, N_2) \tag{168}$$
$$(S_2, N_2) \vdash A_2 \ <: \ C \hookrightarrow (S_3, N_3) \tag{169}$$

By soundness of arrow unification (lemma 38), from equation 167 we get

$$S_1 A = S_1 A_1 \to S_1 A_2 \tag{170}$$
$$S_1 = R_1 \cdot S_0 \tag{171}$$

and $ftv(A_1 \to A_2)$, $vars(S_1)$ are all subset of $vars(S_0) \cup ftv(A) \cup (N_0 - N_1)$, and $ftv(R_1) \subseteq ftv(S_0 A) \cup (N_0 - N_1)$.
Apply induction hypothesis on equation 168 and 169,

$$S_2 B \ <: \ S_2 A_1 \tag{172}$$
$$S_2 = R_2 S_1 \tag{173}$$
$$S_3 A_2 \ <: \ S_3 C \tag{174}$$
$$S_3 = R_3 S_2 \tag{175}$$

with $vars(S_2) \subseteq vars(S_1) \cup ftv(A_1, B) \cup (N_1 - N_2)$, $vars(R_2) \subseteq ftv(S_1 A_1, S_1 B) \cup (N_1 - N_2)$. $vars(S_3) \subseteq vars(S_2) \cup ftv(A_2, C) \cup (N_2 - N_3)$, $vars(R_3) \subseteq ftv(S_2 A_2, S_2 C) \cup (N_2 - N_3)$.
Choose $R = R_3 R_2 R_1$. Combining all the information about free variables, we get that $vars(R) = vars(R_3 R_2 R_1) \subseteq ftv(S_2 A_2, S_2 C) \cup (N_2 - N_3) \cup ftv(S_1 A_1, S_1 B) \cup (N_1 - N_2) \cup ftv(S_0 A) \cup (N_0 - N_1)$. Simplified, we get that $vars(R) \subseteq ftv(S_0 A, S_0 B, S_0 C) \cup \cup (N_0 - N_3)$. Namely, $vars(R) \subseteq ftv(S_0 A, S_0 B, S_0 C) \cup (N_0 - N_3)$. Similarly we can derive the requirement for $vars(S_3)$.
Applying subtyping substitution on equation 172 we could get $R_3 S_2 B \ <: \ R_3 S_2 A_1$, namely $S_3 B \ <: \ S_3 A_1$. Combining with equation 174 and by using subtyping rule for function type, we could get $S_3(A_1 \to A_2) \ <: \ S_3(B \to C)$. In other word, $R_3 R_2 S_1 (A_1 \to A_2) \ <: \ S_3(B \to C)$, namely $R_3 R_2 S_1 A \ <: \ S_3(B \to C)$. So finally $S_3 A \ <: \ S_3(B \to C)$.

- Case AS-FUNL. Similar as case AS-FUNR.
- Case AS-MONO. Holds directly by using soundness of unification and reflexivity of subtyping.

Part 2 We do induction on the subtyping relationship, and analyze each case.

- Case AS-EMPTY. Choose R to be empty, each goal holds trivially.
- Case AS-FORALLL2. Similar to case AS-FORALLL. By induction hypothesis we have

$$S_1 \Psi \vdash S_1(A[\![a \mapsto \widehat{\beta}]\!]) \ <: \ S_1 B \tag{176}$$
$$S_1 = R \cdot S_0 \tag{177}$$

where $vars(S_1)$ and $ftv(B)$ are subsets of $vars(S_0) \cup ftv(\Psi) \cup ftv(A[\![a \mapsto \widehat{\beta}]\!]) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0(A[\![a \mapsto \widehat{\beta}]\!])) \cup ftv(S_0 \Psi) \cup (N_0 - N_1)$.
We can make substitution go inside one step in equation 176 to get

$$S_1 \Psi \vdash (S_1 A)[\![a \mapsto S_1 \widehat{\beta}]\!] \ <: \ S_1 B \tag{178}$$

Our goal $S_1\Psi \vdash S_1(\forall a.A) <: S_1B$ holds by making substitution go inside one step to get $S_1\Psi \vdash \forall a.S_1A <: S_1B$, and applying rule S-ForallL with $S_1\widehat{\beta}$.
Moreover, $vars(S_0) \cup ftv(\Psi) \cup ftv(A[\![a \mapsto \widehat{\beta}]\!]) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(\Psi) \cup ftv(A) \cup \widehat{\beta} \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(\Psi) \cup ftv(A) \cup (N_0\widehat{\beta} - N_1)$. So we get $vars(S_1)$ and $ftv(B)$ are all subsets of $vars(S_0) \cup ftv(\Psi) \cup ftv(A) \cup (N_0\widehat{\beta} - N_1)$. Similarly, we can prove $vars(R) \subseteq ftv(S_0A) \cup ftv(S_0\Psi) \cup (N_0\widehat{\beta} - N_1)$.

– Case AS-FUN2. Given

$$(S_0, N_0) \vdash C <: A \hookrightarrow (S_1, N_1) \tag{179}$$
$$(S_1, N_1) \shortmid \Psi \vdash B <: D \hookrightarrow (S_2, N_2) \tag{180}$$

Applying the first part of soundness of subtyping on equation 179 we get

$$S_1 = R_1S_0 \tag{181}$$
$$S_1C <: S_1A \tag{182}$$

with $vars(S_1) \subseteq vars(S_0) \cup ftv(A, C) \cup (N_0 - N_1)$, $vars(R_1) \subseteq ftv(S_0A, S_0C) \cup (N_0 - N_1)$.
And from the induction hypothesis on equation 180 we get

$$S_2 = R_2S_1 \tag{183}$$
$$S_2\Psi \vdash S_2B <: S_2D \tag{184}$$

with $vars(S_2)$ and $ftv(D)$ subsets of $vars(S_1) \cup ftv(\Psi) \cup ftv(B) \cup (N_1 - N_2)$, $vars(R_2) \subseteq ftv(S_1B) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$. Choose $R = R_2R_1$. Combining all the information about free variables, we get that $vars(R) = vars(R_2R_1) \subseteq ftv(S_1B) \cup ftv(S_1\Psi) \cup (N_1 - N_2) \cup ftv(S_0A, S_0C) \cup (N_0 - N_1)$. Simplified, we get that $vars(R) \subseteq vars(S_0A, S_0B) \cup ftv(S_0(\Psi, C)) \cup (N_0 - N_2)$. Also, $ftv(D)$ and $vars(S_2)$ holds similarly.
By subtyping substitution on equation 182 we get $R_2S_1C <: R_2S_1A$, namely $S_2C <: S_2A$. So we can directly derive $S_2(\Psi, C) \vdash S_2(A \to B) <: S_2(C \to D)$ by using rule S-Fun2.

– Case AS-MONO2. By premises we have

$$(S_0, N_0) \vdash^{\to} \tau = \tau_1 \to \tau_2 \hookrightarrow (S_1, N_1) \tag{185}$$
$$(S_1, N_1) \shortmid \Psi \vdash \tau_1 \to \tau_2 <: B \hookrightarrow (S_2, N_2) \tag{186}$$

By soundness of arrow unification (lemma 38) on equation 185 we get

$$S_1\tau = S_1(\tau_1 \to \tau_2) \tag{187}$$
$$S_1 = R_1 \cdot S_0 \tag{188}$$

Moreover, $ftv(\tau_1 \to \tau_2)$, and $vars(S_1)$ are both subsets of $vars(S_0) \cup ftv(\tau) \cup (N_0 - N_1)$, $vars(R_1) \subseteq ftv(S_0\tau) \cup (N_0 - N_1)$.

By induction hypothesis on equation 186 we get

$$S_2\Psi \vdash S_2(\tau_1 \to \tau_2) \ <: \ S_2 B \tag{189}$$
$$S_2 = R_2 \cdot S_1 \tag{190}$$

Moreover, $vars(S_2)$ and $ftv(B)$ are both subsets of $vars(S_1) \cup ftv(\Psi) \cup ftv(\tau_1 \to \tau_2) \cup (N_1 - N_2)$, $vars(R_2) \subseteq vars(S_1\tau_1, S_1\tau_2) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$.
So we get $S_2\Psi \vdash R_2S_1(\tau_1 \to \tau_2) \ <: \ S_2B$, namely, $S_2\Psi \vdash R_2S_1\tau \ <: \ S_2B$, or $S_2\Psi \vdash S_2\tau \ <: \ S_2B$.
From equation 188 and 190, we get $S_2 = R_2R_1S_0$.
Also, $vars(R_2R_1) \subseteq ftv(S_1\tau_1, S_1\tau_2) \cup ftv(S_1\Psi) \cup (N_1 - N_2) \cup ftv(S_0\tau) \cup (N_0 - N_1) \subseteq ftv(S_0\tau) \cup (S_0\Psi) \cup (N_0 - N_2)$. Similarly, we can prove the request about $vars(S_1)$ and $ftv(B)$.

□

Lemma 40 (Soundness of Generalization). *If* $(S_0, N_0) \vdash \Gamma_{agen}(A) = B \hookrightarrow (S_1, N_1)$, *then* $(S_1\Gamma)_{gen}(S_1A) = S_1B$, *and* $S_1 = S_0$, $ftv(B) \subseteq S_0\Gamma$.

Proof. Do inversion on the generalization relationship, we get that

$$(S_0, N_1\bar{b}) \vdash \Gamma_{agen}(A) = \forall \bar{b}.(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!] \hookrightarrow (S_0, N_1) \tag{191}$$
$$B = \forall \bar{b}.(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!] \tag{192}$$
$$N_0 = N_1\bar{b} \tag{193}$$
$$\overline{\widehat{\alpha}} = ftv(S_0A) - ftv(S_0\Gamma) \tag{194}$$
$$S_1 = S_0 \tag{195}$$

So our goal is to prove $(S_0\Gamma)_{gen}(S_0A) = S_0(\forall \bar{b}.(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!])$. The right hand side equals to $\forall \bar{b}.S_0((S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!])$. Because $(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!]$ is already substituted under S_0, with S_0 well defined and $\bar{b}$ fresh variables, so $S_0((S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!]) = ((S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!])$. Then we can prove $(S_0\Gamma)_{gen}(S_0A) = (\forall \bar{b}.(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!]))$ by α renaming right hand side to $\forall \overline{\widehat{\alpha}}.S_0A$ and by equation 194.

Moreover, $ftv(\forall \bar{b}.(S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!]) = ftv((S_0A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!]) - \{\bar{b}\} = ftv(S_0A) - \overline{\widehat{\alpha}}$. Because of equation 194, $ftv(S_0A) - \overline{\widehat{\alpha}} \subseteq ftv(S_0\Gamma)$. So $ftv(B) \subseteq ftv(S_0\Gamma)$. □

Lemma 41 (Soundness of Typing). *If* $(S_0, N_0) \ ; \ \Gamma \ ; \ \Psi \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$, *then* $S_1\Gamma \ ; \ S_1\Psi \vdash e \ \Rightarrow \ S_1A$. *And* $\exists R$, *that* $S_1 = R \cdot S_0$. *Moreover, both* $vars(S_1)$ *and* $vars(A)$ *are subsets of* $vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_0 - N_1)$.

Proof. By induction on typing relationship, and case analysis.

- Case AT-VAR. By $x : A \in \Gamma$ we have $x : S_1A \in S_1\Gamma$. And by soundness of subtyping we get $S_1\Psi \vdash S_1A \ <: \ S_1B$ with $S_1 = R \cdot S_0$, and $vars(S_1)$, $vars(B)$ all subsets of $ftv(\Psi) \cup ftv(A) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0\Psi) \cup (S_0A)$. Follows directly by using typing rule T-VAR.

– Case AT-INT. Trivially holds by rule T-Int.
– Case AT-LAMANN1. Given

$$(S_0, N_0) \vdash \Gamma, x : A \vdash e \;\Rightarrow\; B \hookrightarrow (S_1, N_1) \tag{196}$$

By induction hypothesis, we have

$$S_1(\Gamma, x : A) \vdash e \;\Rightarrow\; S_1 B \tag{197}$$
$$S_1 = R \cdot S_0 \tag{198}$$

with $vars(S_1)$, $ftv(B)$ both subsets of $vars(S_0) \cup ftv(\Gamma, x : A) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0(\Gamma, x : A)) \cup (N_0 - N_1)$.
Since A is an annotation and $S_1 A = A$, from equation 197, we can derive $S_1\Gamma, x : A \vdash e \;\Rightarrow\; S_1 B$. So we can apply typing rule for annotated lambda and get $S_1\Gamma \vdash \lambda x : A.\ e \;\Rightarrow\; A \to S_1 B$. Again because A is closed and contains no free type variable, so $vars(S_0) \cup ftv(\Gamma, x : A) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(\Gamma) \cup (N_0 - N_1)$, which is superset of $vars(S_1)$, and $ftv(A \to B)$. And $ftv(S_0(\Gamma, x : A)) \cup (N_0 - N_1) = ftv(S_0\Gamma) \cup (N_0 - N_1)$, which is superset of $vars(R)$.
– Case AT-LAMANN2. Given

$$(S_0, N_0) \vdash C \;<:\; A \hookrightarrow (S_1, N_1) \tag{199}$$
$$(S_1, N_1) \vdash \Gamma, x : A \vdash \Psi \vdash e \;\Rightarrow\; B \hookrightarrow (S_2, N_2) \tag{200}$$

Applying soundness of subtyping on equation 199, and from induction hypothesis,

$$S_1 C \;<:\; S_1 A \tag{201}$$
$$S_1 = R_1 S_0 \tag{202}$$
$$S_2(\Gamma, x : A) \vdash S_2\Psi \vdash e \;\Rightarrow\; S_2 B \tag{203}$$
$$S_2 = R_2 S_1 \tag{204}$$

with $vars(S_1)$ subset of $vars(S_0) \cup ftv(A, C) \cup (N_0 - N_1)$, $vars(R_1) \subseteq ftv(S_0 A, S_0 C) \cup (N_0 - N_1)$, $vars(S_2)$ and $ftv(B)$ subsets of $vars(S_1) \cup vars(\Gamma, x : A) \cup vars(\Psi) \cup (N_1 - N_2)$. $vars(R_2) \subseteq vars(S_1(\Gamma, x : A)) \cup vars(S_1\Psi) \cup (N_1 - N_2)$. Choose $R = R_2 R_1$. Since A is an annotation and is closed , so equation 203 can be rewritten as $S_2\Gamma, x : A \vdash S_2\Psi \vdash e \;\Rightarrow\; S_2 B$. Also by subtyping substitution, from equation 201 we get $\vdash R_2 S_1 C \;<:\; R_2 S_1 A$, namely $\vdash S_2 C \;<:\; S_2 A$. Since $S_2 A = A$, so $\vdash S_2 C \;<:\; A$. Then by typing rule for annotated lambda with application context, we get $S_2\Gamma \vdash S_2(\Psi, C) \vdash \lambda x : A.\ e \;\Rightarrow\; S_2 C \to S_2 B$.
Moreover, $vars(R) = vars(R_2 R_1) \subseteq vars(S_0 A, S_0 C) \cup (N_0 - N_1) \cup vars(S_1(\Gamma, x : A)) \cup vars(S_1\Psi) \cup (N_1 - N_2)$. Simplified, $vars(R) \subseteq ftv(S_0\Gamma) \cup ftv(S_0(\Psi, C)) \cup (N_0 - N_2)$, since we can replace S_1 by R_1 and S_0. Similarly, we can prove $vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi, C) \cup (N_0 - N_2)$ is the superset of $vars(C \to B)$ and $vars(S_0)$.

– Case AT-LAM1. Given

$$(S_0, N_0) \shortmid \Gamma, x : \widehat{\beta} \vdash e \;\Rightarrow\; A \hookrightarrow (S_1, N_1) \tag{205}$$

By induction hypothesis, we have

$$S_1(\Gamma, x : \widehat{\beta}) \vdash e \;\Rightarrow\; S_1 A \tag{206}$$
$$S_1 = R S_0 \tag{207}$$

where $vars(S_1)$ and $vars(A)$ subsets of $vars(S_0) \cup ftv(\Gamma, x : \widehat{\beta}) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0(\Gamma, x : \widehat{\beta})) \cup (N_0 - N_1)$. Rewrite equation 206 to $S_1\Gamma, x : S_1\widehat{\beta} \vdash e \;\Rightarrow\; S_1 A$. By typing rule for lambda we get $S_1\Gamma \vdash \lambda x.\ e \;\Rightarrow\; S_1\widehat{\beta} \to S_1 A$.
Moreover, $vars(S_1) \subseteq vars(S_0) \cup ftv(\Gamma, x : \widehat{\beta}) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(\Gamma) \cup (N_0\widehat{\beta} - N_1)$. Similarly, we can prove it is also the superset of and $vars(\widehat{\beta} \to A)$.
Since $\widehat{\beta}$ is fresh to S_0, so $vars(R) \subseteq ftv(S_0\Gamma) \cup (N_0\widehat{\beta} - N_1)$.

– Case AT-LAM2. Given

$$(S_0, N_0) \shortmid \Gamma, x : A \shortmid \Psi \vdash e \;\Rightarrow\; B \hookrightarrow (S_1, N_1) \tag{208}$$

By induction hypothesis, we have

$$S_1(\Gamma, x : A) \shortmid S_1\Psi \vdash e \;\Rightarrow\; S_1 B \tag{209}$$
$$S_1 = R S_0 \tag{210}$$

where $vars(S_1)$ and $vars(B)$ subsets of $vars(S_0) \cup ftv(\Gamma, x : A) \cup ftv(\Psi) \cup (N_0 - N_1)$, $vars(R) \subseteq ftv(S_0(\Gamma, x : A)) \cup ftv(S_0\Psi) \cup (N_0 - N_1)$.
Rewrite equation 209 to $S_1\Gamma, x : S_1 A \shortmid S_1\Psi \vdash e \;\Rightarrow\; S_1 B$. By typing rule for lambda we get $S_1\Gamma \shortmid S_1(\Psi, A) \vdash \lambda x.\ e \;\Rightarrow\; S_1 A \to S_1 B$.
Moreover, $vars(S_1) \subseteq vars(S_0) \cup ftv(\Gamma, x : A) \cup vars(\Psi) \cup (N_0 - N_1) \subseteq vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi, A) \cup (N_0 - N_1)$. Similarly, we can prove it is also the superset of $vars(A \to B)$.
Also, $vars(R) \subseteq ftv(S_0\Gamma) \cup (S_0(\Psi, A)) \cup (N_0 - N_1)$.

– Case AT-APP. Given

$$(S_0, N_0) \shortmid \Gamma \vdash e_2 \;\Rightarrow\; A \hookrightarrow (S_1, N_1) \tag{211}$$
$$(S_1, N_1) \vdash \Gamma_{agen}(A) = B \hookrightarrow (S_2, N_2) \tag{212}$$
$$(S_2, N_2) \shortmid \Gamma \shortmid \Psi, B \vdash e_1 \;\Rightarrow\; B \to C \hookrightarrow (S_3, N_3) \tag{213}$$

By soundness of generalization on equation 212, we get

$$S_2 = S_1 \tag{214}$$
$$(S_2\Gamma)_{gen}(S_2 A) = S_2 B \tag{215}$$

with $ftv(B) \subseteq S_1\Gamma$. So we substitute S_1 for S_2. By induction hypothesis of equation 211 and 213,

$$S_1\Gamma \vdash e_2 \;\Rightarrow\; S_1 A \tag{216}$$
$$S_1 = R_1 S_0 \tag{217}$$
$$S_3\Gamma \shortmid S_3(\Psi, B) \vdash e_1 \;\Rightarrow\; S_3 B \to S_3 C \tag{218}$$
$$S_3 = R_2 S_1 \tag{219}$$

And $vars(S_1)$ and $ftv(A)$ are subsets of $vars(S_0) \cup ftv(\Gamma) \cup (N_0 - N_1)$, $vars(R_1) \subseteq ftv(S_0\Gamma) \cup (N_0 - N_1)$, and $vars(S_3)$ and $ftv(B \to C)$ are subsets of $vars(S_1) \cup ftv(\Gamma) \cup ftv(\Psi, B) \cup (N_2 - N_3)$, and $vars(R_2) \subseteq ftv(S_1\Gamma) \cup ftv(S_1(\Psi, B)) \cup (N_2 - N_3)$.
Choose $R = R_2R_1$, so $S_3 = R_2R_1S_0$. Apply the part 2 of typing substitution lemma on equation 216 and 215, we get for some t,

$$R_2S_1\Gamma \vdash e_2 \;\Rightarrow\; t \tag{220}$$
$$(R_2S_1\Gamma)_{gen}(t) = R_2S_1B \tag{221}$$

Substitute equation 219,

$$S_3\Gamma \vdash e_2 \;\Rightarrow\; t \tag{222}$$
$$(S_3\Gamma)_{gen}(t) = S_3B \tag{223}$$

Use typing rule for application with equation 222, 223 and 218, we can derive $S_3\Gamma \shortmid S_3\Psi \vdash e_1\ e_2 \;\Rightarrow\; S_3C$.
Since $ftv(B) \subseteq ftv(S_1\Gamma) \subseteq vars(S_1) \cup ftv(\Gamma)$, so $vars(S_3) \cup ftv(B \to C) \subseteq vars(S_1) \cup ftv(\Gamma) \cup ftv(\Psi, B) \cup (N_2 - N_3) \subseteq vars(S_0) \cup ftv(\Gamma) \cup (N_0 - N_1) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_2 - N_3) \subseteq vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_0 - N_3)$.
Also, $vars(R_2R_1) \subseteq ftv(S_1\Gamma) \cup (S_1(\Psi, B)) \cup (N_2 - N_3) \cup ftv(S_0\Gamma) \cup (N_0 - N_1)$. Since $vars(S_1) = vars(R_1S_0)$, so we can simplify it to $vars(R_2R_1) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_2 - N_3) \cup ftv(S_0\Gamma) \cup (N_0 - N_1) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_0 - N_3)$.

□

We can then apply a ground substitution V that substitute all the unsolved meta variables to any monotype (like Int) to get the statement of soundness, where the initial application context is also empty.

Theorem 3 (Soundness). *If* $([], N_0) \shortmid \Gamma \vdash e \;\Rightarrow\; A \hookrightarrow (S_1, N_1)$*, then for any substitution* V*, that* $dom(V) = fmv(S_1\Gamma, S_1A)$*, we have* $VS_1\Gamma \vdash e \;\Rightarrow\; VS_1A$*.*

Proof. Follows directly by typing substitution (lemma 35) and typing soundness (lemma 41). □

Completeness

Lemma 42. *if* $S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1$*, then we have:*

1. $vars(S_1) \subseteq vars(S_0) \cup ftv(\tau_1, \tau_2)$
2. $dom(S_0) \subseteq dom(S_1)$
3. $range(S_1) \subseteq range(S_0) \cup ftv(\tau_1, \tau_2)$

Proof. By induction on the unification relation, and it is easy to prove each subgoal. □

Lemma 43. *if* $S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1$, *then* $fov(S_1\tau_1, S_1\tau_2) \subseteq fov(S_0\tau_1, S_0\tau_2)$.

Proof. By induction on the unification relation, and it is easy to prove each subgoal. □

Lemma 44. *if* $(S_0, N_0) \vdash^{\rightarrow} A = A_1 \to A_2 \hookrightarrow (S_1, N_1)$, *then* $fov(S_1A, S_1A_1, S_1A_2) \subseteq fov(S_0A)$.

Proof. By induction on the arrow unification relation, using lemma 43, it is easy to prove each subgoal. □

Lemma 45.

- *if* $(S_0, N_0) \vdash A \mathrel{<:} B \hookrightarrow (S_1, N_1)$, *and* $fov(S_0A, S_0B) \subseteq \chi$, *then* $fov(S_1A, S_1B) \subseteq \chi$.
- *if* $(S_0, N_0) \shortmid \Psi \vdash A \mathrel{<:} B \hookrightarrow (S_1, N_1)$, *and* $fov(S_0A, S_0\Psi) \subseteq \chi$, *then* $fov(S_1A, S_1B, S_1\Psi) \subseteq \chi$.

Proof. This lemma has two parts, with only the second one depending on the first one, so we prove them separately.

Part 1 Do induction on the subtyping relation and case analysis.

- Case AS-FORALLR. We have $fov(S_0A, S_0(\forall a.B)) \in \chi$, given

$$(S_0, N_0) \vdash A \mathrel{<:} B[\![a \mapsto b]\!] \hookrightarrow (S_1, N_1) \tag{224}$$
$$b \notin ftv(S_1A) \tag{225}$$
$$b \notin ftv(S_1(\forall a.B)) \tag{226}$$

  we can derive $fov(S_0A, S_0(B[\![a \mapsto b]\!])) \in \chi b$. So by induction hypothesis, we have $fov(S_1A, S_1(B[\![a \mapsto b]\!])) \in \chi b$. Then because of equation 225 and 226 we have $fov(S_1A, S_1(B[\![a \mapsto b]\!])) \in \chi$. Therefore, $fov(S_1A, S_1(\forall a.B)) \in \chi$.
- Case AS-FORALLL. We have $fov(S_0(\forall a.A), S_0B) \in \chi$, given

$$(S_0, N_0) \vdash A[\![a \mapsto \widehat{\beta}]\!] \mathrel{<:} B \hookrightarrow (S_1, N_1) \tag{227}$$

  Since $\widehat{\beta}$ comes from a fresh name supply, so $\widehat{\beta} \notin vars(S_0)$. So we have $fov(S_0(A[\![a \mapsto \widehat{\beta}]\!]), S_0B) \in \chi$. Then by induction hypothesis, we have $fov(S_1(A[\![a \mapsto \widehat{\beta}]\!]), S_1B) \in \chi$. Therefore, $fov(S_1(\forall a.A), S_1B) \in \chi$.

– Case AS-FUNR. We have $(S_0, N_0) \vdash A \ <: \ B \to C \hookrightarrow (S_3, N_3)$, given

$$(S_0, N_0) \vdash^{\to} A = A_1 \to A_2 \hookrightarrow (S_1, N_1) \tag{228}$$
$$(S_1, N_1) \vdash B \ <: \ A_1 \hookrightarrow (S_2, N_2) \tag{229}$$
$$(S_2, N_2) \vdash A_2 \ <: \ C \hookrightarrow (S_3, N_3) \tag{230}$$

Also, $fov(S_0A, S_0B, S_0C) \in \chi$.
By lemma 44 we have $fov(S_1A, S_1A_1, S_1A_2) \in \chi$.
Claim 1: $fov(S_1B, S_1C) \subseteq \chi$. From soundness of arrow unification (lemma 38), we have

$$S_1 = R_1 \cdot S_0 \tag{231}$$
$$dom(S_1) \in vars(S_0) \cup ftv(A) \cup (N_0 - N_1) \tag{232}$$
$$vars(R_1) \in ftv(S_0A) \cup (N_0 - N_1) \tag{233}$$
$$S_1A = S_1A_1 \to S_1A_2 \tag{234}$$

From $ftv(S_0B, S_0C) \in \chi$, we know that all the ordinary variables in B and C are already in χ. So we consider the meta type variables. If B, C have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(B, C)$. If $\widehat{\alpha} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\alpha} \in dom(S_1)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_0)$. Then consider $\tau = S_0\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_1\widehat{\alpha}) = fov(S_0\widehat{\alpha}) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_0)$. If $\widehat{\beta} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_1)$. Then it must be $\widehat{\beta} \in vars(R_1)$. By equation 233, we have $\widehat{\beta} \in ftv(S_0A) \cup (N_0 - N_1)$. Obviously $\widehat{\beta}$ cannot come from $N_0 - N_1$, since $\widehat{\beta} \in ftv(S_0\widehat{\alpha})$, $\widehat{\alpha} \in ftv(B, C)$, and N_0 is fresh to S_0 and B, C. So $\widehat{\beta} \in ftv(S_0A)$. Therefore, $fov(S_1\widehat{\beta}) \subseteq fov(S_1S_0A) = fov(S_1A) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_0)$. Because we know $\widehat{\alpha} \in dom(S_1)$, then it must be $\widehat{\alpha} \in vars(R_1)$. So $\widehat{\alpha} \in ftv(S_0A) \cup (N_0 - N_1)$. It cannot be $(N_0 - N_1)$ since $\widehat{\alpha} \in ftv(B, C)$ and N_0 is fresh to B, C. So $\widehat{\alpha} \in ftv(S_0A)$. So $fov(S_1\widehat{\alpha}) \subseteq fov(S_1S_0A) = fov(S_1A) \subseteq \chi$.

So we now have $fov(S_1A_1, S_1B) \subseteq \chi$. By induction hypothesis on equation 229 we have $fov(S_2A_1, S_2B) \subseteq \chi$.
Claim 2: $fov(S_2A_2, S_2C) \subseteq \chi$. From soundness of subtyping (lemma 39) on equation 229, we have

$$S_2 = R_2 \cdot S_1 \tag{235}$$
$$dom(S_2) \in vars(S_1) \cup ftv(A_1, B) \cup (N_1 - N_2) \tag{236}$$
$$vars(R_2) \in ftv(S_1A_1, S_1B) \cup (N_1 - N_2) \tag{237}$$

From $fov(S_1A_2, S_1C) \in \chi$, we know that all the ordinary variables in A_2, C are already in χ. So we consider the meta type variables. If A_2, C have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(A_2, C)$. If $\widehat{\alpha} \notin dom(S_2)$ then it is finished. Otherwise $\widehat{\alpha} \in dom(S_2)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_1)$. Then consider $\tau = S_1\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_2\widehat{\alpha}) = fov(S_1\widehat{\alpha}) \in fov(S_1A_2, S_1C) \in \chi$. Otherwise, consider $\widehat{\beta} \in \tau$. Then it must be $\widehat{\beta} \notin dom(S_1)$. If $\widehat{\beta} \notin dom(S_2)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_2)$. Then it must be $\widehat{\beta} \in vars(R_2)$. By equation 237, we have $\widehat{\beta} \in ftv(S_1A_1, S_1B) \cup (N_1 - N_2)$. Obviously $\widehat{\beta}$ cannot come from $(N_1 - N_2)$, since $\widehat{\beta} \in (S_1\widehat{\alpha})$, and $\widehat{\alpha} \in ftv(A_2, C)$, and N_1 is fresh to S_1 and A_2, C. So $\widehat{\beta} \in ftv(S_1A_1, S_1B)$. So $fov(S_2\widehat{\beta}) \subseteq fov(S_2S_1A_1, S_2S_1B) = fov(S_2A_1, S_2B) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_1)$. Because we know $\widehat{\alpha} \in dom(S_2)$, then it must be $\widehat{\alpha} \in vars(R_2)$. So $\widehat{\alpha} \in ftv(S_1A_1, S_1B) \cup (N_1 - N_2)$. It cannot be $(N_1 - N_2)$ since $\widehat{\alpha} \in ftv(A_2, C)$ and N_1 is fresh to A_2, C. So $\widehat{\alpha} \in ftv(S_1A_1, S_1B)$. So $fov(S_2\widehat{\alpha}) \subseteq fov(S_2S_1A_1, S_2S_1B) = fov(S_2A_1, S_2B) \subseteq \chi$.

So we now have $fov(S_2A_2, S_2C) \subseteq \chi$. By induction hypothesis on equation 230 we have $fov(S_3A_2, S_3C) \subseteq \chi$.

Claim 3: $fov(S_3A_1, S_3B) \in \chi$.

From soundness of subtyping on equation 230, we have

$$S_3 = R_3 \cdot S_2 \tag{238}$$

$$dom(S_3) \in vars(S_2) \cup ftv(A_2, C) \cup (N_2 - N_3) \tag{239}$$

$$vars(R_3) \in ftv(S_2A_2, S_2C) \cup (N_2 - N_3) \tag{240}$$

From $fov(S_2A_1, S_2B) \in \chi$, we know that all the ordinary variables in A_1, B are already in χ. So we consider the meta type variables. If A_1, B have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(A_1, B)$. If $\widehat{\alpha} \notin dom(S_3)$ then it is finished. Otherwise $\widehat{\alpha} \in dom(S_3)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_2)$. Then consider $\tau = S_2\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_3\widehat{\alpha}) = fov(S_2\widehat{\alpha}) \in fov(S_2A_1, S_2B) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_2)$. If $\widehat{\beta} \notin dom(S_3)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_3)$. Then it must be $\widehat{\beta} \in vars(R_3)$. By equation 240, we have $\widehat{\beta} \in ftv(S_2A_2, S_2C) \cup (N_2 - N_3)$. Obviously $\widehat{\beta}$ cannot come from $N_2 - N_3$, since $\widehat{\beta} \in ftv(S_2\widehat{\alpha})$, and $\widehat{\alpha} \in ftv(A_1, B)$, and N_2 is fresh to S_2 and A_1, B. So $\widehat{\beta} \in ftv(S_2A_2, S_2C)$. So $fov(S_3\widehat{\beta}) \subseteq fov(S_3S_2A_2, S_3S_2C) = fov(S_3A_2, S_3C) \in \chi$.
- $\widehat{\alpha} \notin dom(S_2)$. Because we know $\widehat{\alpha} \in dom(S_3)$, then it must be $\widehat{\alpha} \in vars(R_3)$. So $\widehat{\alpha} \in ftv(S_2A_2, S_2C) \cup (N_2 - N_3)$. It cannot be $(N_2 - N_3)$ since $\widehat{\alpha} \in ftv(A_1, B)$ and N_2 is fresh to A_1, B. So $\widehat{\alpha} \in ftv(S_2A_2, S_2C)$. So $fov(S_3\widehat{\alpha}) \subseteq fov(S_3S_2A_2, S_3S_2C) = fov(S_3A_2, S_3C) \subseteq \chi$.

So we now have $fov(S_3A_1, S_3A_2, S_3B, S_3C) \subseteq \chi$. From equation 234 it is easy to derive $S_3A = S_3A_1 \to S_3A_2$. So $fov(S_3A, S_3B, S_3C) \in \chi$ and we are done.

– Case AS-FUNL. Similar as last case.
– Case AS-MONO. Follows directly by lemma 43.

Part 2 Do induction on the subtyping relation and case analysis.

– Case AS-EMPTY. Follows trivially.
– Case AS-FORALLL2. We have$(S_0, N_0\widehat{\beta}) \shortmid \Psi \vdash \forall a.A \ <: \ B \hookrightarrow (S_1, N_1)$, given

$$(S_0, N_0) \shortmid \Psi \vdash A[\![a \mapsto \widehat{\beta}]\!] \ <: \ B \hookrightarrow (S_1, N_1) \tag{241}$$

And $fov(S_0(\forall a.A)) \cup fov(S_0\Psi) \in \chi$. We will assume a is a fresh name. So by induction hypothesis, we have $fov(S_1\Psi) \cup fov(S_1(A[\![a \mapsto \widehat{\beta}]\!])) \cup fov(S_1B) \subseteq fov(S_0\Psi) \cup fov(S_0(A[\![a \mapsto \widehat{\beta}]\!]))$. We know $\widehat{\beta} \notin dom(S_0)$ since $\widehat{\beta}$ comes from a fresh name supply. So $fov(S_1\Psi) \cup fov(S_1(A[\![a \mapsto \widehat{\beta}]\!])) \cup fov(S_1B) \subseteq fov(S_0\Psi) \cup fov(S_0A) - \{a\} = fov(S_0\Psi) \cup fov(S_0(\forall a.A)) \subseteq \chi$. Namely, $fov(S_1\Psi) \cup fov(S_1(\forall a.A)) \cup fov(S_1\widehat{\beta}) \cup fov(S_1B) \subseteq \chi$. So we are done.
– Case AS-FUN2. We have $(S_0, N_0) \shortmid \Psi, C \vdash A \to B \ <: \ C \to D \hookrightarrow (S_2, N_2)$, given

$$(S_0, N_0) \vdash C \ <: \ A \hookrightarrow (S_1, N_1) \tag{242}$$
$$(S_1, N_1) \shortmid \Psi \vdash B \ <: \ D \hookrightarrow (S_2, N_2) \tag{243}$$

And $fov(S_0A, S_0B, S_0\Psi, S_0C) \in \chi$. Apply part 1 of this lemma to equation 242, we get $fov(S_1A, S_1C) \subseteq \chi$. Apply induction hypothesis on equation 243, we have $fov(S_1B, S_1D, S_1\Psi) \subseteq \chi$. So the goal is proved.
– Case AS-MONO2. We have $(S_0, N_0) \shortmid \Psi \vdash \tau \ <: \ B \hookrightarrow (S_2, N_2)$, given

$$(S_0, N_0) \vdash^{\to} \tau = \tau_1 \to \tau_2 \hookrightarrow (S_1, N_1) \tag{244}$$
$$(S_1, N_1) \shortmid \Psi \vdash \tau_1 \to \tau_2 \ <: \ B \hookrightarrow (S_2, N_2) \tag{245}$$

And $fov(S_0\Psi, S_0\tau) \in \chi$.
Apply lemma 44 to equation 244, we have $fov(S_1\tau, S_1\tau_1, S_1\tau_2) \in \chi$. By soundness of arrow unification, we have

$$S_1\tau = S_1\tau_1 \to S_1\tau_2 \tag{246}$$
$$S_1 = R_1 \cdot S_0 \tag{247}$$
$$vars(R_1) \subseteq ftv(S_0\tau) \cup (N_0 - N_1) \tag{248}$$

Claim 1: $fov(S_1\Psi) \in \chi$.
From $fov(S_0\Psi) \in \chi$, we know that all the ordinary variables in Ψ are already in χ. So we consider the meta type variables. If Ψ have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in vars(\Psi)$. If $\widehat{\alpha} \notin dom(S_1)$ then it is finished. Otherwise $\widehat{\alpha} \in dom(S_1)$.
Consider two cases:

- $\widehat{\alpha} \in dom(S_0)$. Then consider $\tau = S_0\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_1\widehat{\alpha}) = fov(S_0\widehat{\alpha}) \in fov(S_0\Psi) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_0)$. If $\widehat{\beta} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_1)$. Then it must be $\widehat{\beta} \in vars(R_1)$. By equation 248 we have $\widehat{\beta} \in ftv(S_0\tau) \cup (N_0 - N_1)$. Obviously $\widehat{\beta}$ cannot come from $(N_0 - N_1)$, since $\widehat{\beta} \in ftv(S_0\widehat{\alpha})$, $\widehat{\alpha} \in vars(\Psi)$, and N_0 is fresh to Ψ. So $\widehat{\beta} \in ftv(S_0\tau)$. So $fov(S_1\widehat{\beta}) \subseteq fov(S_1S_0\tau) = fov(S_1\tau) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_0)$. Because we know $\widehat{\alpha} \in dom(S_1)$, then it must be $\widehat{\alpha} \in vars(R_1)$. So $\widehat{\alpha} \in ftv(S_0\tau) \cup (N_0 - N_1)$. It cannot be $(N_0 - N_1)$ since $\widehat{\alpha} \in vars(\Psi)$ and N_0 is fresh to Ψ. So $\widehat{\alpha} \in ftv(S_0\tau)$. So $fov(S_1\widehat{\alpha}) \subseteq fov(S_1S_0\tau) = fov(S_1\tau) \subseteq \chi$.

Now we have $fov(S_1\Psi, S_1\tau_1, S_1\tau_2) \in \chi$. By induction hypothesis on equation 245 we have $fov(S_2\tau_1, S_2\tau_2, S_2\Psi, S_2B) \in \chi$. Also it is easy to derive $S_2\tau = S_2\tau_1 \to S_2\tau_2$. So $fov(S_2\tau, S_2\Psi, S_2B) \subseteq \chi$, which is exactly what we want, so we are done.

□

Lemma 46. *if* $(S_0, N_0) \vdash \Gamma \vdash \Psi \vdash e \Rightarrow A \hookrightarrow (S_1, N_1)$*, and* $fov(S_0\Gamma, S_0\Psi) \subseteq \chi$*, then* $fov(S_1\Gamma, S_1\Psi, S_1A) \subseteq \chi$*.*

Proof. Do induction on the typing relationship and case analysis.

– Case AT-VAR. We have $(S_0, N_0) \vdash \Gamma \vdash \Psi \vdash x \Rightarrow B \hookrightarrow (S_1, N_1)$, given

$$x : A \in \Gamma \tag{249}$$

$$(S_0, N_0) \vdash \Psi \vdash A <: B \hookrightarrow (S_1, N_1) \tag{250}$$

And $fov(S_0\Gamma, S_0\Psi) \subseteq \chi$. Because $x : A \in \Gamma$, so $fov(S_0A) \subseteq \chi$. Then by Lemma 45, we have $fov(S_1A, S_1B, S_1\Psi) \subseteq \chi$.
Now we want to prove $fov(S_1\Gamma) \subseteq \chi$. By soundness of subtyping, we have

$$S_1 = R \cdot S_0 \tag{251}$$

$$vars(R) \subseteq ftv(S_0A) \cup ftv(S_0\Psi) \cup (N_0 - N_1) \tag{252}$$

From $fov(S_0\Psi) \subseteq \chi$, we know that all the ordinary variables in Ψ are already in χ. So we consider the meta type variables. If Ψ have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(\Psi)$. If $\widehat{\alpha} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\alpha} \in dom(S_1)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_0)$. Then consider $\tau = S_0\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_1\widehat{\alpha}) = fov(S_0\widehat{\alpha}) \subseteq \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_0)$. If $\widehat{\beta} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_1)$. Then it must be $\widehat{\beta} \in vars(R)$. By equation 252, we have $\widehat{\beta} \in ftv(S_0A) \cup ftv(S_0\Psi) \cup (N_0 - N_1)$. Obviously $\widehat{\beta}$ cannot come from $(N_0 - N_1)$, since $\widehat{\beta} \in (S_0\widehat{\alpha})$, $\widehat{\alpha} \in ftv(\Psi)$, and N_0 is fresh to S_0 and Ψ. So $\widehat{\beta} \in ftv(S_0A) \cup ftv(S_0\Psi)$. Therefore, $fov(S_1\widehat{\beta}) \subseteq fov(S_1S_0A) \cup ftv(S_1S_0\Psi) = fov(S_1A) \cup fov(S_1\Psi) \subseteq \chi$

- $\widehat{\alpha} \notin dom(S_0)$. Because we know $\widehat{\alpha} \in dom(S_1)$, then it must be $\widehat{\alpha} \in vars(R)$. So $\widehat{\alpha} \in ftv(S_0 A) \cup ftv(S_0 sctx) \cup (N_0 - N_1)$. It cannot be $(N_0 - N_1)$ since $\widehat{\alpha} \in ftv(\Psi)$, and N_0 is fresh to Ψ. So $\widehat{\alpha} \in ftv(S_0 A) \cup ftv(S_0 \Psi)$. So $fov(S_1 \widehat{\alpha}) \subseteq fov(S_1 S_0 A) \cup ftv(S_1 S_0 \Psi) = fov(S_1 A) \cup ftv(S_1 \Psi) \subseteq \chi$.

– Case AT-INT. Follows directly.

– Case AT-LAMANN1. We have $(S_0, N_0) \vdash \Gamma \vdash \lambda x : A.\ e \Rightarrow A \to B \hookrightarrow (S_1, N_1)$, given

$$(S_0, N_0) \vdash \Gamma, x : A \vdash e \Rightarrow B \hookrightarrow (S_1, N_1) \tag{253}$$

And $fov(S_0 \Gamma) \in \chi$.
Because A is an annotation and is closed, so $fov(S_0(\Gamma, x : A)) \in \chi$. Then by induction hypothesis, we have $fov(S_1(\Gamma, x : A)) \cup fov(S_1 B) \in \chi$. So $fov(S_1 \Gamma) \cup fov(S_1(A \to B)) \in \chi$.

– Case AT-LAMANN2. We have $(S_0, N_0) \vdash \Gamma \vdash \Psi, C \vdash \lambda x : A.\ e \Rightarrow C \to B \hookrightarrow (S_2, N_2)$, given

$$(S_0, N_0) \vdash C <: A \hookrightarrow (S_1, N_1) \tag{254}$$

$$(S_1, N_1) \vdash \Gamma, x : A \vdash \Psi \vdash e \Rightarrow B \hookrightarrow (S_2, N_2) \tag{255}$$

And $fov(S_0 \Gamma) \cup fov(S_0(\Psi, C)) \subseteq \chi$.
Because A is closed, so we have $fov(S_0 A, S_0 C) \subseteq \chi$. By Lemma 45 on equation 254, we have $fov(S_1 A, S_1 C) \subseteq \chi$.
Claim 1: $fov(S_1 \Gamma) \cup fov(S_1 \Psi) \subseteq \chi$.
From soundness of subtyping on equation 254 we have

$$S_1 = R_1 \cdot S_0 \tag{256}$$

$$vars(R_1) \subseteq ftv(S_0 C) \cup ftv(S_0 A) \cup (N_0 - N_1) \tag{257}$$

From $fov(S_0 \Gamma) \cup fov(S_0 \Psi) \in \chi$, we know that all the ordinary variables in Γ, Ψ are already in χ. So we consider the meta type variables. If Γ, Ψ have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(\Gamma, \Psi)$. If $\widehat{\alpha} \notin dom(S_1)$, then it is finished. Otherwise, $\widehat{\alpha} \in dom(S_1)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_0)$. Then consider $\tau = S_0 \widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_1 \widehat{\alpha}) = fov(S_0 \widehat{\alpha}) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_0)$ If $\widehat{\beta} \notin dom(S_1)$ then it is finished. Otherwise $\widehat{\beta} \in dom(S_1)$. Then it must be $\widehat{\beta} \in vars(R_1)$. By equation 257, we have $\widehat{\beta} \in ftv(S_0 C) \cup ftv(S_0 A) \cup (N_0 - N_1)$. Obviously $\widehat{\beta}$ cannot come from $(N_0 - N_1)$, since $\widehat{\beta} \in ftv(S_0 \widehat{\alpha})$, $\widehat{\alpha} \in ftv(\Gamma, \Psi)$, and N_0 is fresh to S_0 and Γ, Ψ. So $\widehat{\beta} \in ftv(S_0 C) \cup ftv(S_0 A)$. Therefore, $fov(S_1 \widehat{\beta}) \subseteq fov(S_1 S_0 C) \cup ftv(S_1 S_0 A) = ftv(S_1 C) \cup ftv(S_1 A) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_0)$. Because we know $\widehat{\alpha} \in dom(S_1)$, then it must be $\widehat{\alpha} \in vars(R_1)$. So $\widehat{\alpha} \in ftv(S_0 C) \cup ftv(S_0 A) \cup (N_0 - N_1)$. It cannot be $(N_0 - N_1)$ since $\widehat{\alpha} \in ftv(\Psi, \Gamma)$ and N_0 is fresh to Ψ, Γ. So $fov(S_1 \widehat{\alpha}) \subseteq fov(S_1 S_0 A, S_1 S_0 C) = fov(S_1 A, S_1 C) \subseteq \chi$.

Now we have $fov(S_1(\Gamma, x : A)) \cup fov(S_1\Psi) \subseteq \chi$, by induction hypothesis on equation 255 we have $fov(S_2(\Gamma, x : A)) \cup fov(S_2\Psi) \cup fov(S_2 B) \in \chi$.
Claim 2: $fov(S_2 C) \subseteq \chi$.
From soundness of typing on equation 255 we have

$$S_2 = R_2 \cdot S_1 \tag{258}$$
$$vars(R_2) \subseteq ftv(S_1(\Gamma, x : A)) \cup ftv(S_1\Psi) \cup (N_1 - N_2) \tag{259}$$

From $fov(S_1 C) \in \chi$, we know that all the ordinary variables in C are already in χ. So we consider the meta type variables. If C have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(C)$. If $\widehat{\alpha} \notin dom(S_2)$, then it is finished. Otherwise, $\widehat{\alpha} \in dom(S_2)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_1)$. Then consider $\tau = S_1\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_2\widehat{\alpha}) = fov(S_1\widehat{\alpha}) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_1)$. If $\widehat{\beta} \notin dom(S_2)$ then it is finished. Otherwise $\widehat{\beta} \in dom(S_2)$. Then it must be $\widehat{\beta} \in vars(R_2)$. by equation 259, we have $\widehat{\beta} \in ftv(S_1\Gamma) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$. Obviously $\widehat{\beta}$ cannot come from $(S_1 - N_2)$, since $\widehat{\beta} \in ftv(S_1\widehat{\alpha})$, $\widehat{\alpha} \in ftv(C)$, and N_1 is fresh to S_1 and C. So $\widehat{\beta} \in ftv(S_1\Gamma) \cup ftv(S_1\Psi)$. Therefore, $fov(S_2\widehat{\beta}) \subseteq fov(S_2S_1\Gamma) \cup ftv(S_2S_1\Psi) = fov(S_2 tctx) \cup fov(as_2\Psi) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_1)$. Because we know $\widehat{\alpha} \in dom(S_2)$, then it must be $\widehat{\alpha} \in vars(R_2)$. So $\widehat{\alpha} \in ftv(S_1\Gamma) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$. It cannot be $(N_1 - N_2)$ since $\widehat{\alpha} \in ftv(C)$ and N_1 is fresh to C. So $fov(S_2\widehat{\alpha}) \subseteq fov(S_2S_1\Gamma, S_2S_1\Psi) = fov(S_2\Gamma, S_2\Psi) \subseteq \chi$.

Therefore $fov(S_2\Gamma) \cup fov(S_2(\Psi, C)) \cup fov(S_2(C \to B)) \in \chi$ and we are done.

– Case AT-LAM1. We have $(S_0, N_0\widehat{\beta}) \shortmid \Gamma \vdash \lambda x.\ e \Rightarrow \widehat{\beta} \to A \hookrightarrow (S_1, N_1)$, given

$$(S_0, N_0) \shortmid \Gamma, x : \widehat{\beta} \vdash e \Rightarrow A \hookrightarrow (S_1, N_1) \tag{260}$$

And $fov(S_0\Gamma) \in \chi$. Because $\widehat{\beta}$ comes from a fresh name supply so $\widehat{\beta} \notin S_0$. So $fov(S_0(\Gamma, x : \widehat{\beta})) \in \chi$. Then by induction hypothesis, we have $fov(S_1(\Gamma, x : \widehat{\beta})) \cup fov(S_1 A) \in \chi$, namely $fov(S_1\Gamma) \cup fov(S_1(\widehat{\beta} \to A)) \in \chi$.

– Case AT-LAM2. We have $(S_0, N_0) \shortmid \Gamma \shortmid \Psi, A \vdash \lambda x.\ e \Rightarrow A \to B \hookrightarrow (S_1, N_1)$, given

$$(S_0, N_0) \shortmid \Gamma, x : A \shortmid \Psi \vdash e \Rightarrow B \hookrightarrow (S_1, N_1) \tag{261}$$

And $fov(S_0\Gamma) \cup fov(S_0(\Psi, A)) \in \chi$. Namely, $fov(S_0(\Gamma, x : A)) \cup fov(S_0\Psi) \in \chi$. So by induction hypothesis, we have $fov(S_1(\Gamma, x : A)) \cup fov(S_1\Psi) \cup (S_1 B) \in \chi$ and we are done.

– Case AT-APP. We have$(S_0, N_0) \shortmid \Gamma \shortmid \Psi \vdash e_1\ e_2 \Rightarrow C \hookrightarrow (S_3, N_3)$, given

$$(S_0, N_0) \shortmid \Gamma \vdash e_2 \Rightarrow A \hookrightarrow (S_1, N_1) \tag{262}$$
$$(S_1, N_1) \vdash \Gamma_{agen}(A) = B \hookrightarrow (S_2, N_2) \tag{263}$$
$$(S_2, N_2) \shortmid \Gamma \shortmid \Psi, B \vdash e_1 \Rightarrow B \to C \hookrightarrow (S_3, N_3) \tag{264}$$

And $fov(S_0\Gamma) \cup fov(S_0\Psi) \in \chi$.
By inversion on equation 263, we have

$$S_2 = S_1 \tag{265}$$

$$B = \forall \bar{b}.(S_1 A)[\![\overline{\widehat{\alpha}} \mapsto \bar{b}]\!] \tag{266}$$

By induction hypothesis on equation 262, we have $fov(S_1\Gamma) \cup fov(S_1 A) \in \chi$. So it is easy to derive that $fov(S_1 B) \in \chi$.
Claim 1: $fov(S_1\Psi) \in \chi$.
By soundness of typing on equation 262, we have

$$S_1 = R_1 \cdot S_0 \tag{267}$$

$$vars(R_1) \subseteq ftv(S_0\Gamma) \cup (N_0 - N_1) \tag{268}$$

From $fov(S_0\Psi) \in \chi$, we know that all the ordinary variables in Ψ are already in χ. So we consider the meta type variables. If Ψ have no meta type variables then it is finished; otherwise, consider $\widehat{\alpha} \in ftv(\Psi)$. If $\widehat{\alpha} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\alpha} \in dom(S_1)$. Consider two cases:

- $\widehat{\alpha} \in dom(S_0)$. Then consider $\tau = S_0\widehat{\alpha}$. If τ contains no meta type variables, then it is finished because $fov(S_1\widehat{\alpha}) = fov(S_0\widehat{\alpha}) \in \chi$. Otherwise, consider $\widehat{\beta} \in ftv(\tau)$. Then it must be $\widehat{\beta} \notin dom(S_0)$. If $\widehat{\beta} \notin dom(S_1)$ then it is finished. Otherwise, $\widehat{\beta} \in dom(S_1)$. Then it must be $\widehat{\beta} \in vars(R_1)$. By equation 268, we have $\widehat{\beta} \in ftv(S_0\Gamma) \cup (N_0 - N_1)$. Obviously $\widehat{\beta}$ cannot come from $N_0 - N_1$, since $\widehat{\beta} \in ftv(S_0\widehat{\alpha})$, $\widehat{\alpha} \in ftv(\Psi)$, and N_0 is fresh to S_0 and Ψ. So $\widehat{\beta} \in ftv(S_0\Gamma)$. Therefore, $fov(S_1\widehat{\beta}) \subseteq fov(S_1 S_0\Gamma) = fov(S_1\Gamma) \subseteq \chi$.
- $\widehat{\alpha} \notin dom(S_0)$. Because we know $\widehat{\alpha} \in dom(S_1)$, then it must be $\widehat{\alpha} \in vars(R_1)$. So $\widehat{\alpha} \in ftv(S_0\Gamma) \cup (N_0 - N_1)$. It cannot be $(N_0 - N_1)$ since $\widehat{\alpha} \in ftv(\Psi)$ and N_0 is fresh to Ψ. So $\widehat{\alpha} \in ftv(S_0\Gamma)$. So $fov(S_1\widehat{\alpha}) \subseteq fov(S_1 S_0\Gamma) = fov(S_1\Gamma) \subseteq \chi$.

So we now have $fov(S_1\Gamma) \cup fov(S_1(\Psi, B)) \in \chi$. Since $S_1 = S_2$, by induction hypothesis on equation 264, we have $fov(S_3\Gamma) \cup fov(S_3(\Psi, B)) \cup fov(S_3(B \to C)) \in \chi$ and we are done

□

Lemma 47 (Completeness of Unification). *if $SS_0\tau_1 = SS_0\tau_2$, then $\exists S_1$, that $S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1$. And $\exists R$, that $S \cdot S_0 = R \cdot S_1$. Moreover, $vars(R) \subseteq vars(S_0) \cup vars(S) \cup ftv(\tau_1, \tau_2)$.*

Proof. We do induction on following lexicographic pair

$$\langle |range(S_0) \cup ftv(\tau_1, \tau_2)|, size(\tau_1) + size(\tau_2) \rangle$$

and proceed by case analysis on the possible forms of τ_1 and τ_2.

– τ_1, τ_2 are both arrow types. So $\tau_1 = \tau_{11} \to \tau_{12}$, $\tau_2 = \tau_{21} \to \tau_{22}$. By inversion we have

$$SS_0\tau_{11} = SS_0\tau_{21} \tag{269}$$
$$SS_0\tau_{12} = SS_0\tau_{22} \tag{270}$$

Since

$$\begin{aligned}\langle|range(S_0) \cup ftv(\tau_{11}, \tau_{21})|, size(\tau_{11}, \tau_{21})\rangle < \\ \langle|range(S_0) \cup ftv(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22})|, size(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22})\rangle\end{aligned} \tag{271}$$

By induction on equation 269, we have

$$S_0 \vdash \tau_{11} = \tau_{21} \hookrightarrow S_1 \tag{272}$$
$$SS_0 = R_1S_1 \tag{273}$$

Moreover, $vars(R_1) \subseteq vars(S) \cup vars(S_0) \cup ftv(\tau_{11}, \tau_{21})$.
And by soundness of unification (lemma 37) on equation 272, we have $vars(S_1) \subseteq vars(S_0) \cup ftv(\tau_{11}, \tau_{21})$.
From $SS_0\tau_{12} = SS_0\tau_{22}$, with equation 273 we have $R_1S_1\tau_{12} = R_1S_1\tau_{22}$.
By lemma 42 that $range(S_1) \subseteq range(S_0) \cup ftv(\tau_{11}, \tau_{21})$. So we have

$$\begin{aligned}\langle|range(S_1) \cup ftv(\tau_{12}, \tau_{22})|, size(\tau_{12}, \tau_{22})\rangle < \\ \langle|range(S_0) \cup ftv(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22})|, size(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22})\rangle\end{aligned} \tag{274}$$

then by induction, we have

$$S_1 \vdash \tau_{12} = \tau_{22} \hookrightarrow S_2 \tag{275}$$
$$R_1S_1 = R_2S_2 \tag{276}$$

And $vars(R_2) \subseteq vars(R_1) \cup vars(S_1) \cup ftv(\tau_{12}, \tau_{22})$.
So, feed S_2 and R_2 to the goal, we have $S_1 \vdash \tau_1 = \tau_2 \hookrightarrow S_2$ by using AU-Fun with equation 272 and 275. And $SS_0 = R_2S_2$ by transitivity with equation 273 and 276.

– Case they are both ordinary variables. Then they must be the same ordinary variable and we can prove the case trivially by using AU-Refl.
– Case they are both Int, holds trivially.
– Case when one is a meta variables. Without loss of generality, we assume $\tau_1 = \widehat{\alpha}$. There are several cases:
  • $\tau_2 = \widehat{\alpha}$. Then we can prove the case trivially by AU-Refl. So in the following cases, we will assume $\tau_2 \neq \widehat{\alpha}$.
  • $\widehat{\alpha} \in dom(S_0)$. We have $SS_0\widehat{\alpha} = SS_0\tau_2$, or equivalently $SS_0(S_0\widehat{\alpha}) = SS_0\tau_2$. Since we know S_0 is well-defined, and $\widehat{\alpha} \in dom(S_0)$, so $\widehat{\alpha} \notin range(S_0)$. And obviously $\widehat{\alpha} \notin \tau_2$ since otherwise $SS_0\widehat{\alpha} = SS_0\tau_2$ cannot hold. So we have

$$|range(S_0) \cup ftv(S_0\widehat{\alpha}, \tau)| < |range(S_0) \cup ftv(\widehat{\alpha}, \tau)| \tag{277}$$

because $ftv(S_0\widehat{\alpha}) \in range(S_0)$, also we know the left hand side contains $\widehat{\alpha}$ but the right hand side does not.
Therefore we can apply the induction hypothesis to get

$$S_0 \vdash S_0\widehat{\alpha} = \tau_2 \hookrightarrow S_1 \tag{278}$$
$$SS_0 = RS_1 \tag{279}$$

With the same S_1 and R, by AU-BVAR1 we are done.

- $\widehat{\alpha} \notin dom(S_0)$. Then we have following sub-cases:
  * $S_0\tau = \widehat{\alpha}$. Since $\tau \neq \widehat{\alpha}$, we have that $\tau = \widehat{\beta}$ and $S_0\widehat{\beta} = \widehat{\alpha}$. Also because $S_0\widehat{\alpha} = \widehat{\alpha}$, then the goal $S_0 \vdash \widehat{\alpha} = \widehat{\beta} \hookrightarrow S_0$ holds by using AU-BVAR1 with $S_0 \vdash \widehat{\alpha} = \widehat{\alpha} \hookrightarrow S_0$. Take $R = S$ we are done. So in the following cases we assume $S_0\tau \neq \widehat{\alpha}$.
  * $\widehat{\alpha} \in ftv(S_0\tau)$. Impossible case because then $SS_0\widehat{\alpha} = SS_0\tau$ becomes $S\widehat{\alpha} = S(S_0\tau)$ and the sizes of types in both side are not equal now.
  * $\widehat{\alpha} \notin ftv(S_0\tau)$. In this case we apply AU-VAR1 to get the goal $S_0 \vdash \widehat{\alpha} = \tau \hookrightarrow [\![\widehat{\alpha} \mapsto S_0\tau]\!] \cdot S_0$. Then we need to give the R makes $SS_0 = R \cdot [\![\widehat{\alpha} \mapsto S_0\tau]\!]S_0$. We already know $SS_0\widehat{\alpha} = S\widehat{\alpha} = SS_0\tau$. We can rewrite $S = S' \cdot [\![\widehat{\alpha} \mapsto S_0\tau]\!]$. So choose $R = S'$ and we are done. □

Lemma 48 (Completeness of Arrow Unification). *If $SS_0A = B \to C$, then for any fresh name supply N_0, we have for some S_1, $(S_0, N_0) \vdash^{\to} A = A_1 \to A_2 \hookrightarrow (S_1, N_1)$, and $\exists R$, that $S \cdot S_0 = R \cdot S_1\backslash_{N_0 - N_1}$, and $RS_1A_1 = B$, $RS_1A_2 = C$. Moreover, $vars(R) \subseteq vars(S) \cup vars(S_0) \cup ftv(A) \cup (N_0 - N_1)$.*

Proof. By case analysis on the type structure of A.

- A is Int or an ordinary variable. Impossible.
- A is a meta variable $\widehat{\alpha}$. Then both B and C are monotypes. Then we would like to use rule AF-MONO, where we have $(S_0, N_0\widehat{\alpha}_1\widehat{\alpha}_2) \vdash^{\to} \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow (S_1, N_0)$, if given $S_0 \vdash \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1$. So our goal now is to prove the later one. There are two sub-cases:
  * $\widehat{\alpha} \notin dom(S_0)$. Here we are going to use AU-VAR1, because definitely $\widehat{\alpha} \notin ftv(S_0(\widehat{\alpha}_1 \to \widehat{\alpha}_2)) = \{\widehat{\alpha}_1, \widehat{\alpha}_2\}$ since $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ come from a fresh name supply. So $S_1 = [\![\widehat{\alpha} \mapsto \widehat{\alpha}_1 \to \widehat{\alpha}_2]\!] \cdot S_0$.
    Then in order to prove $S \cdot S_0 = R \cdot S_1\backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\alpha} \mapsto \widehat{\alpha}_1 \to \widehat{\alpha}_2]\!] \cdot S_0\backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, pick $R = S \cdot [\![\widehat{\alpha}_1 \mapsto B]\!] \cdot [\![\widehat{\alpha}_2 \mapsto C]\!]$.
  * $\widehat{\alpha} \in dom(S_0)$. Then we case analyze the form of $S_0\widehat{\alpha}$
    * $S_0\widehat{\alpha}$ is Int or an ordinary variable. Impossible.
    * $S_0\widehat{\alpha} = \widehat{\beta}$ where $\widehat{\beta} \neq \widehat{\alpha}$. Also because S_0 is well-defined, so $\widehat{\beta} \notin dom(S_0)$. Then in order to give $S_0 \vdash \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1$, we are going to use rule AU-BVAR1 and turn to give $S_0 \vdash S_0\widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1$, namely $S_0 \vdash \widehat{\beta} = (\widehat{\alpha}_1 \to \widehat{\alpha}_2) \hookrightarrow S_1$. For this one, we are going to use rule AU-VAR1 to conclude $S_1 = [\![\widehat{\beta} \mapsto \widehat{\alpha}_1 \to \widehat{\alpha}_2]\!]S_0$.
      In order to prove $S \cdot S_0 = R \cdot S_1\backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\beta} \mapsto \widehat{\alpha}_1 \to \widehat{\alpha}_2]\!] \cdot S_0\backslash_{\widehat{\alpha}_1, \widehat{\alpha}_1}$, we pick $R = S \cdot [\![\widehat{\alpha}_1 \mapsto B]\!] \cdot [\![\widehat{\alpha}_2 \mapsto C]\!]$.

  - $S_0\widehat{\alpha} = D_1 \to D_2$. Then in order to give $S_0 \vdash \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1$, we are going to use rule AU-BVAR1 and turn to give $S_0 \vdash D_1 \to D_2 = \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1$. We assume D_1 and D_2 are not meta type variables, because the cases when they are meta variables are similar. Then we can apply rule AU-FUN and with a result $S_1 = [\![\widehat{\alpha}_2 \mapsto D_2]\!] \cdot [\![\widehat{\alpha}_1 \mapsto D_1]\!] \cdot S_0$.
    In order to prove $S \cdot S_0 = R \cdot S_1 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\alpha}_2 \mapsto D_2]\!] \cdot [\![\widehat{\alpha}_1 \mapsto D_1]\!] \cdot S_0 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, we choose $R = S$. It is easy to derive this holds so we are done.

– A is a function type so $A = D_1 \to D_2$. So $SS_0D_1 = B$ and $SS_0D_2 = C$. Then by AF-ARROW we have $(S_0, N_0) \vdash^{\to} D_1 \to D_2 = D_1 \to D_2 \hookrightarrow (S_0, N_0)$, namely $A_1 = D_1$ and $A_2 = D_2$. Then choose $R = S$ and we are done.

□

Lemma 49 (Completeness of Subtyping).

1. *If $SS_0A <: SS_0B$, then for any fresh name supply N_0, we have for some S_1, that $(S_0, N_0) \vdash A <: B \hookrightarrow (S_1, N_1)$, and $\exists R$, that $S \cdot S_0 = R \cdot S_1 \backslash_{N_0 - N_1}$. Moreover, N_1 is fresh to R.*
2. *If $SS_0\Psi \vdash SS_0A <: B$, then for any fresh name supply N_0, we have for some S_1, that $(S_0, N_0) \shortmid \Psi \vdash A <: C \hookrightarrow (S_1, N_1)$, and $\exists R$, that $S \cdot S_0 = R \cdot S_1 \backslash_{N_0 - N_1}$. And $RS_1C = B$. Moreover, N_1 is fresh to R.*

Proof. This lemma has two parts, with only the second one depending on the first one. So we prove them separately.

Part 1 By induction on the height of derivation. And we case analyze the last rule used in the derivation

– Case S-Int. This means $SS_0A = SS_0B = \mathsf{Int}$. Both A and B are mono types.
  From the completeness of unification (lemma 47), we know for some S_1, that $S_0 \vdash A = B \hookrightarrow S_1$. And $S \cdot S_0 = R \cdot S_1$, with $vars(R) \subseteq vars(S_0) \cup vars(S) \cup ftv(A, B)$. So by using rule AS-MONO we have $(S_0, N_0) \vdash A <: B \hookrightarrow (S_1, N_0)$. Choose the same R and we are done.

– Case S-VAR. Similar to the case S-INT.

– Case S-FORALLR. So $SS_0B = \forall a.C$ and $SS_0A <: \forall a.C$. Suppose a is fresh because we can always use α renaming to choose a fresh one. Then we know $B = \forall a.B_1$ since $\forall$s are preserved by substitution. Also $SS_0B_1 = C$. Suppose b is one fresh name provided by name supply, by α renaming, $\forall a.C = \forall b.C[\![a \mapsto b]\!]$. So $SS_0A <: \forall b.C[\![a \mapsto b]\!]$. Then according to rule S-FORALLR we have $SS_0A <: C[\![a \mapsto b]\!]$. Equivalently, $SS_0A <: SS_0(B_1[\![a \mapsto b]\!])$. By induction hypothesis

$$(S_0, N_0 b) \vdash A <: B_1[\![a \mapsto b]\!] \hookrightarrow (S_1, N_1) \tag{280}$$

  And $SS_0 = RS_1 \backslash_{N_0 b - N_1}$.
  Then we can apply rule AS-FORALLR to get $(S_0, N_0 b) \vdash A <: \forall a.B_1 \hookrightarrow (S_1, N_1)$, namely $(S_0, N_0 b) \vdash A <: B \hookrightarrow (S_1, N_1)$ if we can satisfy other

premises, which are $b \notin ftv(S_1 A)$ and $b \notin ftv(S_1 B)$. We already know $b \notin ftv(A, B)$ because b comes from a fresh name supply. Then suppose by contradiction that $b \in ftv(S_1 A, S_1 B)$. There must be a $\widehat{\beta} \in ftv(A, B)$ and $b \in ftv(S_1 \widehat{\beta})$. Since $\widehat{\beta} \notin N_0 b - N_1$ since name supply is fresh, we have $SS_0\widehat{\beta} = RS_1\widehat{\beta}$. Then we have $b \in ftv(RS_1\widehat{\beta}) = ftv(SS_0\widehat{\beta})$. So $b \in vars(S, S_0)$. But b is a fresh name to S, S_0, a contradiction.

– Case S-Fun. Then we have $SS_0 A = A_1 \to A_2$, and $SS_0 B = B_1 \to B_2$. There are both two cases for both A and B:

- A is a meta type variable $A = \widehat{\alpha}$ and B is a meta variable $B = \widehat{\beta}$. We have $SS_0\widehat{\alpha} <: SS_0\widehat{\beta}$. Because both sides are monotypes, so it must be $SS_0\widehat{\alpha} = SS_0\widehat{\beta}$. Then by completeness of unification (lemma 47), we have

$$S_0 \vdash \widehat{\alpha} = \widehat{\beta} \hookrightarrow S_1 \tag{281}$$
$$SS_0 = RS_1 \tag{282}$$
$$vars(R) \subseteq vars(S_0) \cup vars(S) \cup ftv(\widehat{\alpha}, \widehat{\beta}) \tag{283}$$

  Then by rule AS-Mono we have $(S_0, N_0) \vdash \widehat{\alpha} <: \widehat{\beta} \hookrightarrow (S_1, N_0)$. All the subgoals follow directly.
- A is a meta variable and B is a function type $B = D_1 \to D_2$, and $SS_0 D_1 = B_1$, $SS_0 D_2 = B_2$.
  Because $SS_0 A = A_1 \to A_2$, by completeness of arrow unification (lemma 48), we have

$$(S_0, N_0) \vdash^{\to} A = C_1 \to C_2 \hookrightarrow (S_1, N_1) \tag{284}$$
$$SS_0 = R_1 S_1 \backslash_{N_0 - N_1} \tag{285}$$
$$R_1 S_1 C_1 = A_1 \tag{286}$$
$$R_1 S_1 C_2 = A_2 \tag{287}$$

  Moreover, N_1 is fresh to R_1.
  From the preconditions, we have $B_1 <: A_1$, namely $SS_0 D_1 <: R_1 S_1 C_1$, equivalently, $R_1 S_1 D_1 <: R_1 S_1 C_1$, because $ftv(D_1)$ has no variables in $(N_0 - N_1)$. From this, we can apply induction hypothesis:

$$(S_1, N_1) \vdash D_1 <: C_1 \hookrightarrow (S_2, N_2) \tag{288}$$
$$R_2 S_2 = R_1 S_1 \backslash_{N_1 - N_2} \tag{289}$$

  Moreover, N_2 is fresh to R_2.
  Similarly, from hypothesis we have $A_2 <: B_2$, namely $R_1 S_1 C_2 <: SS_0 D_2$, equivalently $R_2 S_2 C_2 <: R_2 S_2 D_2$. By induction hypothesis we have

$$(S_2, N_2) \vdash C_2 <: D_2 \hookrightarrow (S_3, N_3) \tag{290}$$
$$R_3 S_3 = R_2 S_2 \backslash_{N_2 - N_3} \tag{291}$$

  Moreover, N_3 is fresh to R_3.

Then by rule AS-FunR with equation 284, 288 and 290, we can derive $(S_0, N_0) \vdash A \mathrel{<:} D_1 \to D_2 \hookrightarrow (S_3, N_3)$, namely $(S_0, N_0) \vdash A \mathrel{<:} B \hookrightarrow (S_3, N_3)$. And from equation 285, 289 and 291 we get $SS_0 = R_3 S_3 \backslash_{N_0 - N_3}$.

- A is function type and B is meta variable. Similar to last case.
- both A and B are function types. Similar to last case.

– Case S-FORALLL. Then $A = \forall a.A_1$. Assume $a \notin vars(S, S_0)$. Because we can always by α renaming achieve this requirement. Then we have $\forall a.SS_0A_1 \mathrel{<:} SS_0B$ given that $[\![a \mapsto \tau]\!]SS_0A_1 \mathrel{<:} SS_0B$, where $a \notin ftv(\tau)$. Consider a $\widehat{\beta}$ comes from a fresh name supply $N_0\widehat{\beta}$, then we have $[\![\widehat{\beta} \mapsto \tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} SS_0B$. Equivalently, $[\![\widehat{\beta} \mapsto \tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} [\![\widehat{\beta} \mapsto \tau]\!]SS_0B$.
However, τ may contain variables that in N_0. We know N_0 is fresh to SS_0A_1 and SS_0B. So we only need to take care of the type variables in τ but not in $ftv(SS_0A_1, SS_0B)$. Consider a renaming substitution Q that maps those type variables to fresh variables. By subtyping substitution, we have $[\![\widehat{\beta} \mapsto Q\tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} [\![\widehat{\beta} \mapsto Q\tau]\!]SS_0B$. Now by induction hypothesis we have

$$(S_0, N_0) \vdash A_1[\![a \mapsto \widehat{\beta}]\!] \mathrel{<:} B \hookrightarrow (S_1, N_1) \tag{292}$$

$$RS_1 = [\![\widehat{\beta} \mapsto Q\tau]\!]SS_0 \backslash_{N_0 - N_1} \tag{293}$$

Moreover, N_1 is fresh to R.
Then from equation 292 by AS-Forall we have $(S_0, N_0\widehat{\beta}) \vdash \forall a.A_1 \mathrel{<:} B \hookrightarrow (S_1, N_1)$.
From equation 293 we have $RS_1 = SS_0 \backslash_{N_0\widehat{\beta} - N_1}$

Part 2 By induction on the height of derivation. And we case analyze the last rule used in the derivation

– Case S-EMPTY. So $\Psi = \emptyset$, $SS_0A = B$. By AS-Empty and choose $R = S$, each subgoal holds trivially.

– Case S-FORALLL. Then $A = \forall a.A_1$. Assume $a \notin vars(S, S_0)$. Because we can always by α renaming achieve this requirement. Then we have $SS_0\Psi \vdash \forall a.SS_0A_1 \mathrel{<:} B$ given that $SS_0\Psi \vdash [\![a \mapsto \tau]\!]SS_0A_1 \mathrel{<:} B$, where $a \notin ftv(\tau)$. Consider a $\widehat{\beta}$ comes from a fresh name supply $N_0\widehat{\beta}$, then we have $SS_0\Psi \vdash [\![\widehat{\beta} \mapsto \tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} B$. Equivalently, $[\![\widehat{\beta} \mapsto \tau]\!]SS_0\Psi \vdash [\![\widehat{\beta} \mapsto \tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} B$.
However, τ may contain variables that in N_0. We know N_0 is fresh to SS_0A_1 and B. So we only need to take care of the type variables in τ but not in $ftv(SS_0A_1, B)$. Consider a renaming substitution Q that maps those type variables to fresh variables. By subtyping substitution, we have $[\![\widehat{\beta} \mapsto Q\tau]\!]SS_0\Psi \vdash [\![\widehat{\beta} \mapsto Q\tau]\!]SS_0(A_1[\![a \mapsto \widehat{\beta}]\!]) \mathrel{<:} B$. Now by induction hypothesis we have

$$(S_0, N_0) \shortmid \Psi \vdash A_1[\![a \mapsto \widehat{\beta}]\!] \mathrel{<:} C \hookrightarrow (S_1, N_1) \tag{294}$$

$$RS_1 = [\![\widehat{\beta} \mapsto Q\tau]\!]SS_0 \backslash_{N_0 - N_1} \tag{295}$$

$$RS_1C = B \tag{296}$$

Moreover, N_1 is fresh to R.
Then from equation 294 by AS-FORALL2 we have $(S_0, N_0\widehat{\beta}) \shortmid \Psi \vdash \forall a.A_1 \ <: \ C \hookrightarrow (S_1, N_1)$.
From equation 295 we have $RS_1 = SS_0 \backslash_{N_0\widehat{\beta}-N_1}$

– Case S-Fun2. Then we have $SS_0A = A_1 \to A_2$, and $B = B_1 \to B_2$. Assume $\Psi = \Psi', D_1$, then $SS_0D_1 = B_1$.
There are two cases for A:

- A is a meta variable.
  Because $SS_0A = A_1 \to A_2$, by completeness of arrow unification (lemma 48), we have

$$(S_0, N_0) \vdash^{\to} A = C_1 \to C_2 \hookrightarrow (S_1, N_1) \tag{297}$$
$$SS_0 = R_1S_1 \backslash_{N_0-N_1} \tag{298}$$
$$R_1S_1C_1 = A_1 \tag{299}$$
$$R_1S_1C_2 = A_2 \tag{300}$$

  Moreover, N_1 is fresh to R_1.
  From the premises, we have $B_1 \ <: \ A_1$, namely $SS_0D_1 \ <: \ R_1S_1C_1$, equivalently, $R_1S_1D_1 \ <: \ R_1S_1C_1$, because $ftv(D_1)$ has no variables in $(N_0 - N_1)$. From this, we can apply the part 1 of this lemma to get:

$$(S_1, N_1) \vdash D_1 \ <: \ C_1 \hookrightarrow (S_2, N_2) \tag{301}$$
$$R_2S_2 = R_1S_1 \backslash_{N_1-N_2} \tag{302}$$

  Moreover, N_2 is fresh to R_2.
  From hypothesis we have $SS_0\Psi' \vdash A_2 \ <: \ B_2$, namely $R_1S_1\Psi' \vdash R_1S_1C_2 \ <: \ B_2$, equivalently $R_2S_2\Psi' \vdash R_2S_2C_2 \ <: \ B_2$. By induction hypothesis we have

$$(S_2, N_2) \shortmid \Psi' \vdash C_2 \ <: \ D_2 \hookrightarrow (S_3, N_3) \tag{303}$$
$$R_3S_3 = R_2S_2 \backslash_{N_2-N_3} \tag{304}$$
$$R_3S_3D_2 = B_2 \tag{305}$$

  Moreover, N_3 is fresh to R_3.
  Then by rule AS-FUN2 with equation 301, 303, we can derive $(S_1, N_1) \shortmid \Psi', D_1 \vdash C_1 \to C_2 \ <: \ D_1 \to D_2 \hookrightarrow (S_3, N_3)$, namely $(S_1, N_1) \shortmid \Psi \vdash C_1 \to C_2 \ <: \ D_1 \to D_2 \hookrightarrow (S_3, N_3)$.
  And with AS-MONO2 and equation 297, we have $(S_0, N_0) \shortmid \Psi \vdash A \ <: \ D_1 \to D_2 \hookrightarrow (S_3, N_3)$.
  And from equation 298, 302 and 304 we get $SS_0 = R_3S_3 \backslash_{N_0-N_3}$.
  $(R_3S_3)(D_1 \to D_2) = R_3S_3D_1 \to R_3S_3D_2 = SS_0D_1 \to B_2 = B_1 \to B_2$.
- A is function type. Then we have $A = C_1 \to C_2$, and $SS_0C_1 = A_1$, $SS_0C_2 = A_2$.

From the premises, we have $B_1 <: A_1$, namely $SS_0D_1 <: SS_0C_1$, applying the part 1 of this lemma we get

$$(S_0, N_0) \vdash D_1 <: C_1 \hookrightarrow (S_1, N_1) \tag{306}$$
$$SS_0 = R_1S_1\backslash_{N_0-N_1} \tag{307}$$

Moreover, N_1 is fresh to R_1.
From the premises we have $SS_0\Psi' \vdash A_2 <: B_2$, namely $R_1S_1\Psi' \vdash R_1S_1C_2 <: B_2$. By induction hypothesis we have

$$(S_1, N_1) \shortmid \Psi' \vdash C_2 <: D_2 \hookrightarrow (S_2, N_2) \tag{308}$$
$$R_2S_2 = R_1S_1\backslash_{N_1-N_2} \tag{309}$$
$$R_2S_2D_2 = B_2 \tag{310}$$

Moreover, N_2 is fresh to R_2.
Then by rule AS-Fun2 with equation 306, 308, we can derive $(S_0, N_0) \shortmid \Psi', D_1 \vdash C_1 \to C_2 <: D_1 \to D_2 \hookrightarrow (S_2, N_2)$, namely $(S_0, N_0) \shortmid \Psi \vdash A <: D_1 \to D_2 \hookrightarrow (S_2, N_2)$.
And from equation 307, 309 we get $SS_0 = R_2S_2\backslash_{N_0-N_2}$.
$(R_2S_2)(D_1 \to D_2) = R_2S_2D_1 \to R_2S_2D_2 = SS_0D_1 \to B_2 = B_1 \to B_2$.

□

Before we state the completeness of typing, let us consider an example $(\lambda x.\ x)\ (\lambda x.\ x)$. In algorithmic system, the result type is deterministic, which is $\forall a.a \to a$. However in declarative system, it is possible to derive the type $\mathsf{Int} \to \mathsf{Int}$. For the type $\mathsf{Int} \to \mathsf{Int}$, it is impossible to use substitution to make two result types equivalent. Namely, the result type come from algorithmic system is possibly more polymorphic than the one from declarative system. But with this statement, the completeness of typing involving generalization becomes hard to prove. So instead of stating the subsumption between the result type, we state the subsumption between the generalized result types.

Lemma 50 (Completeness of Typing).

1. *If $SS_0\Gamma \shortmid SS_0\Psi \vdash e \Rightarrow A$, and $ftv(SS_0\Psi) \subseteq ftv(SS_0\Gamma)$, then for a fresh name supply N_0, $\exists S_1, N_1, B$, that $(S_0, N_0) \shortmid \Gamma \shortmid \Psi \vdash e \Rightarrow B \hookrightarrow (S_1, N_1)$, and $\exists R$, that $S \cdot S_0 = R \cdot S_1\backslash_{N_0-N_1}$, Also, $\overline{SS_0\Gamma(RS_1B)} <: \overline{SS_0\Gamma(A)}$. Moreover, N_1 is fresh.*
2. *If $SS_0\Gamma \vdash e \Rightarrow A_1$, and $SS_0\Gamma_{gen}(A_1) = A_2$, then for a fresh name supply N_0, that $(S_0, N_0) \shortmid \Gamma \vdash e \Rightarrow B_1 \hookrightarrow (S_1, N_1)$, and $(S_1, N_1) \vdash \Gamma_{agen}(B_1) = B_2 \hookrightarrow (S_1, N_2)$ and $\exists R$, that $S \cdot S_0 = R \cdot S_1\backslash_{N_0-N_2}$, and $RS_1B_2 <: A_2$. Moreover, N_2 is fresh.*

Proof. This lemma has two parts and they relies on each other. So we prove them simultaneously.

Part 1 By induction on the height of derivation, and case analyze the last rule used in the derivation.

– Case T-VAR. We have $e = x$, and $x : C \in SS_0\Gamma$, and $SS_0\Psi \vdash C \ <: \ A$. So there must be $x : B \in \Gamma$, and $SS_0 B = C$. Therefore, $SS_0\Psi \vdash SS_0 B \ <: \ A$. By completeness of subtyping (lemma 49) we have

$$(S_0, N_0) \vdash \Psi \vdash B \ <: \ D \hookrightarrow (S_1, N_1) \tag{311}$$
$$S \cdot S_0 = R \cdot S_1 \backslash_{N_0 - N_1} \tag{312}$$
$$RS_1 D = A \tag{313}$$

So by rule AT-VAR, we have $(S_0, N_0) \vdash \Gamma \vdash \Psi \vdash e \ \Rightarrow \ D \hookrightarrow (S_1, N_1)$. And all other goals follow naturally.

– Case T-INT. Choose $R = S$ and all subgoals are trivial.

– Case T-LAMANN. we have $\Psi = \emptyset$, $A = B \to C$, and $SS_0\Gamma \vdash \lambda x : B. \ e_1 \ \Rightarrow \ B \to C$, given $SS_0\Gamma, x : B \vdash e_1 \ \Rightarrow \ C$. Because B is a user defined type so it is closed. So we can rewrite the equation as $SS_0(\Gamma, x : B) \vdash e_1 \ \Rightarrow \ C$. By induction hypothesis, we have

$$(S_0, N_0) \vdash \Gamma, x : B \vdash e_1 \ \Rightarrow \ D \hookrightarrow (S_1, N_1) \tag{314}$$
$$S \cdot S_0 = R \cdot S_1 \backslash_{N_0 - N_1} \tag{315}$$
$$\overline{SS_0(\Gamma, x : B)(RS_1 D)} \ <: \ \overline{SS_0(\Gamma, x : B)(C)} \tag{316}$$

By equation 314 and rule AT-LAMANN1 we have $(S_0, N_0) \vdash \Gamma \vdash \lambda x : B. \ e_1 \ \Rightarrow \ B \to D \hookrightarrow (S_1, N_1)$. Because B is closed, equation 316 can be rewritten as $\overline{SS_0\Gamma(RS_1 D)} \ <: \ \overline{SS_0\Gamma(C)}$.
Then we have $\overline{SS_0\Gamma(RS_1(B \to D))} = \overline{SS_0\Gamma(B \to RS_1 D)} \ <: \ \overline{SS_0\Gamma(B \to C)}$.

– Case T-LAMANN2. we have $A = A_1 \to A_2$, and assume $\Psi = \Psi', C$, then $SS_0 C = A_1$. From premises, we have $SS_0\Gamma \vdash SS_0\Psi', SS_0 C \vdash \lambda x : B. \ e_1 \ \Rightarrow \ A_1 \to A_2$, given $SS_0\Gamma, x : B \vdash SS_0\Psi' \vdash e_1 \ \Rightarrow \ A_2$, and $SS_0 C \ <: \ B$. Because B is a user defined type, so equivalently $SS_0 C \ <: \ SS_0 B$.
By completeness of subtyping (lemma 49), we have

$$(S_0, N_0) \vdash C \ <: \ B \hookrightarrow (S_1, N_1) \tag{317}$$
$$S \cdot S_0 = R_1 \cdot S_1 \backslash_{N_0 - N_1} \tag{318}$$

with N_1 fresh.
Again because B is a user defined type and so is closed, we can rewrite the typing premise as $SS_0(\Gamma, x : B) \vdash SS_0\Psi' \vdash e_1 \ \Rightarrow \ A_2$. Equivalently, $R_1 S_1(\Gamma, x : B) \vdash R_1 S_1 \Psi' \vdash e_1 \ \Rightarrow \ A_2$. By induction hypothesis, we have

$$(S_1, N_1) \vdash \Gamma, x : B \vdash \Psi' \vdash e_1 \ \Rightarrow \ D_2 \hookrightarrow (S_2, N_2) \tag{319}$$
$$R_1 \cdot S_1 = R_2 \cdot S_2 \backslash_{N_1 - N_2} \tag{320}$$
$$\overline{SS_0(\Gamma, x : B)(R_2 S_2 D_2)} \ <: \ \overline{SS_0(\Gamma, x : B)(A_2)} \tag{321}$$

with N_2 fresh.
By rule AT-LAMANN2 with equation 317 and 319 we have $(S_0, N_0) \vdash \Gamma \vdash \Psi', C \vdash \lambda x : B. \ e_1 \ \Rightarrow \ C \to D_2 \hookrightarrow (S_2, N_2)$.

From equation 318 and 320 we get $SS_0 = R_2S_2\backslash_{N_0-N_2}$.
Since B is an annotation and is closed, so equation 321 can be rewritten as $\overline{SS_0\Gamma(R_2S_2D_2)} <: \overline{SS_0\Gamma(A_2)}$.
By precondition we have $ftv(SS_0C) \subseteq ftv(SS_0\Gamma)$. So we can deduce $\overline{SS_0\Gamma(R_2S_2C \to R_2S_2D_2)} = \overline{SS_0\Gamma(SS_0C \to R_2S_2D_2)} <: \overline{SS_0\Gamma(SS_0C \to A_2)} = \overline{SS_0\Gamma(A_1 \to A_2)}$

– Case T-LAM. So $A = A_1 \to A_2$. Assume $\Psi = \Psi', C$, then $SS_0C = A_1$. We have $SS_0\Gamma \mid SS_0\Psi', SS_0C \vdash \lambda x.\ e_1 \Rightarrow A_1 \to A_2$, given $SS_0\Gamma, x : SS_0C \mid SS_0\Psi' \vdash e_1 \Rightarrow A_2$. Equivalently, $SS_0(\Gamma, x : C) \mid SS_0\Psi' \vdash e_1 \Rightarrow A_2$. By induction hypothesis, we have

$$(S_0, N_0) \mid \Gamma, x : C \mid \Psi' \vdash e_1 \Rightarrow D_2 \hookrightarrow (S_1, N_1) \tag{322}$$
$$S \cdot S_0 = R \cdot S_1\backslash_{N_0-N_1} \tag{323}$$
$$\overline{SS_0(\Gamma, x : C)(RS_1D_2)} <: \overline{SS_0(\Gamma, x : C)(A_2)} \tag{324}$$

with N_1 fresh.
So from equation 322 and rule AT-LAM2 we get $(S_0, N_0) \mid \Gamma \mid \Psi', C \vdash \lambda x.\ e_1 \Rightarrow C \to D_2 \hookrightarrow (S_1, N_1)$.
From preconditions, we have $ftv(SS_0C) \subseteq ftv(SS_0\Gamma)$, so equation 324 can be rewritten as $\overline{SS_0\Gamma(RS_1D_2)} <: \overline{SS_0\Gamma(A_2)}$.
Also, $\overline{SS_0\Gamma(RS_1C \to RS_1D_2)} = \overline{SS_0\Gamma(SS_0C \to RS_1D_2)} <: \overline{SS_0\Gamma(SS_0C \to A_2)} = \overline{SS_0\Gamma(A_1 \to A_2)}$.

– Case T-LAM2. So $\Psi = \emptyset$, and $A = \tau \to A_2$. We have $SS_0\Gamma \vdash \lambda x.\ e_1 \Rightarrow \tau \to A_2$, given $SS_0\Gamma, x : \tau \vdash e_1 \Rightarrow A_2$. Suppose $\widehat{\beta}$ comes from a fresh name supply $N_0\widehat{\beta}$, then we have $[\![\widehat{\beta} \mapsto \tau]\!]SS_0(\Gamma, x : \widehat{\beta}) \vdash e_1 \Rightarrow A_2$. By induction hypothesis, we have

$$(S_0, N_0) \mid \Gamma, x : \widehat{\beta} \vdash e_1 \Rightarrow C_2 \hookrightarrow (S_1, N_1) \tag{325}$$
$$[\![\widehat{\beta} \mapsto \tau]\!]S \cdot S_0 = R \cdot S_1\backslash_{N_0-N_1} \tag{326}$$
$$\overline{[\![\widehat{\beta} \mapsto \tau]\!]SS_0(\Gamma, x : \widehat{\beta})(RS_1C_2)} <: \overline{[\![\widehat{\beta} \mapsto \tau]\!]SS_0(\Gamma, x : \widehat{\beta})(A_2)} \tag{327}$$

with N_1 fresh.
From equation 325 and rule AT-LAM1, we have $(S_0, N_0\widehat{\beta}) \mid \Gamma \vdash \lambda x.\ e_1 \Rightarrow \widehat{\beta} \to C_2 \hookrightarrow (S_1, N_1)$.
From equation 326 we have $S \cdot S_0 = R \cdot S_1\backslash_{N_0\widehat{\beta}-N_1}$.
And equation 327 can be rewritten as $\overline{(SS_0\Gamma, x : \tau)(RS_1C_2)} <: \overline{(SS_0\Gamma, x : \tau)(A_2)}$.
Suppose it is $\forall\bar{a}.RS_1C_2 <: \forall\bar{b}.A_2$, where $\bar{a} = ftv(RS_1C_2) - ftv(SS_0\Gamma) - ftv(\tau)$, $\bar{b} = ftv(A_2) - ftv(SS_0\Gamma) - ftv(\tau)$. What we want is $\overline{SS_0\Gamma(RS_1(\widehat{\beta} \to C_2))} <: \overline{SS_0\Gamma(\tau \to A_2)}$, namely $\overline{SS_0\Gamma(\tau \to RS_1C_2)} <: \overline{SS_0\Gamma(\tau \to A_2)}$.
Suppose $\bar{c} = ftv(\tau) - ftv(SS_0\Gamma)$, then what we want is $\forall\bar{c}\bar{a}.\tau \to RS_1C_1 <: \forall\bar{c}\bar{b}.\tau \to A_2$, equivalently $\forall\bar{a}.\tau \to RS_1C_1 <: \forall\bar{b}.\tau \to A_2$ which can be easily derived from $\forall\bar{a}.RS_1C_2 <: \forall\bar{b}.A_2$.

– Case T-APP. From premises, we have

$$SS_0\Gamma \vdash e_2 \;\Rightarrow\; B_1 \tag{328}$$
$$SS_0\Gamma_{gen}(B_1) = B_2 \tag{329}$$
$$SS_0\Gamma \mid SS_0\Psi, B_2 \vdash e_1 \;\Rightarrow\; B_2 \to A \tag{330}$$

Suppose $\chi = ftv(B_2) - vars(S, S_0) \cup ftv(\Gamma, \Psi) \cup ftv(A)$. Consider a substitution Q maps the variables in χ to a fresh set. Then according to the substitution lemma (lemma 35), we have

$$SS_0\Gamma \vdash e_2 \;\Rightarrow\; B_3 \tag{331}$$
$$SS_0\Gamma_{gen}(B_3) = QB_2 \tag{332}$$
$$SS_0\Gamma \mid SS_0\Psi, QB_2 \vdash e_1 \;\Rightarrow\; QB_2 \to A \tag{333}$$

From part 2 of this lemma on equation 331 and 332,

$$(S_0, N_0) \mid \Gamma \vdash e_2 \;\Rightarrow\; B_4 \hookrightarrow (S_1, N_1) \tag{334}$$
$$(S_1, N_1) \vdash \Gamma_{agen}(B_4) = B_5 \hookrightarrow (S_1, N_2) \tag{335}$$
$$S \cdot S_0 = R_1 \cdot S_1 \backslash_{N_0 - N_2} \tag{336}$$
$$R_1 S_1 B_5 \;<:\; QB_2 \tag{337}$$

Because the application context decides the form of typing result, we can derive from equation 330 that

$$A = SS_0\Psi \to A' \tag{338}$$
$$SS_0\Gamma \mid SS_0\Psi, B_2 \vdash e_1 \;\Rightarrow\; B_2 \to SS_0\Psi \to A' \tag{339}$$

According to lemma 33, with equation 339 and 337, we have

$$SS_0\Gamma \mid SS_0\Psi, R_1S_1B_5 \vdash e_1 \;\Rightarrow\; R_1S_1B_5 \to SS_0\Psi \to A'_2 \tag{340}$$
$$\overline{SS_0\Gamma(A'_2)} \;<:\; \overline{SS_0\Gamma(A')} \tag{341}$$

Let $A_2 = SS_0\Psi \to A'_2$. From preconditions we know $ftv(SS_0\Psi) \subseteq ftv(SS_0\Gamma)$, so from equation 341, we can derive $\overline{SS_0\Gamma(SS_0\Psi \to A'_2)} \;<:\; \overline{SS_0\Gamma(SS_0\Psi \to A')}$, namely $\overline{SS_0\Gamma(A_2)} \;<:\; \overline{SS_0\Gamma(A)}$.
Rewrite equation 340 we get $R_1S_1\Gamma \mid R_1S_1\Psi, R_1S_1B_5 \vdash e_1 \;\Rightarrow\; R_1S_1B_5 \to A_2$. Apply the induction hypothesis, (since application context decides the form of result type, so instead of A_3 we use $B_5 \to A_3$)

$$(S_1, N_2) \mid \Gamma \mid R_1, B_5 \vdash e_1 \;\Rightarrow\; B_5 \to A_3 \hookrightarrow (S_3, N_3) \tag{342}$$
$$R_2S_3 = R_1S_1\backslash_{N_2 - N_3} \tag{343}$$
$$\overline{R_1S_1\Gamma(R_2S_3(B_5 \to A_3))} \;<:\; \overline{R_1S_1\Gamma(R_1S_1B_5 \to A_2)} \tag{344}$$

Combining equation 334, 335, 342 with rule AT-APP, we get $(S_0, N_0) \mid \Gamma \mid \Psi \vdash e_1\ e_2 \;\Rightarrow\; A_3 \hookrightarrow (S_3, N_3)$.

From equation 336 and 343 we get $S \cdot S_0 = R_2 \cdot S_3 \backslash_{N_0 - N_3}$.
We can rewrite equation 344 as $\overline{R_1 S_1 \Gamma(R_1 S_1 B_5 \to R_2 S_3 A_3)} <: \overline{R_1 S_1 \Gamma(R_1 S_1 B_5 \to A_2)}$.
Because B_5 is generalized under Γ, so $ftv(B_5) <: ftv(\Gamma)$, so $ftv(R_1 S_1 B_5) \subseteq ftv(R_1 S_1 \Gamma)$. Then we can derive $\overline{R_1 S_1 \Gamma(R_2 S_3 A_3)} <: \overline{R_1 S_1 \Gamma(A_2)}$, namely $\overline{S S_0 \Gamma(R_2 S_3 A_3)} <: \overline{S S_0 \Gamma(A_2)}$. By transitivity, $\overline{S S_0 \Gamma(R_2 S_3 A_3)} <: \overline{S S_0 \Gamma(A)}$.

Part 2 From preconditions, we have

$$S S_0 \Gamma \vdash e \Rightarrow A_1 \tag{345}$$
$$S S_0 \Gamma_{gen}(A_1) = A_2 \tag{346}$$

with $A_2 = \forall \bar{a}. A_1$, where $\bar{a} = ftv(A_1) - ftv(S S_0 \Gamma)$.
Apply part 1 of this lemma on the equation 345, we get

$$(S_0, N_0) \shortmid \Gamma \vdash e \Rightarrow B_1 \hookrightarrow (S_1, N_1) \tag{347}$$
$$R \cdot S_1 = S \cdot S_0 \backslash_{N_0 - N_1} \tag{348}$$
$$\overline{S S_0 \Gamma(R S_1 B_1)} <: \overline{S S_0 \Gamma(A_1)} \tag{349}$$

From equation 348, we have $R \cdot S_1 = S \cdot S_0 \backslash_{N_0 - N_2}$.
In order to be able to use rule AT-Gen, we need to show that $ftv(S_1 B_1) - ftv(S_1 \Gamma)$ are all meta type variables, which means $\Gamma_{agen}(B_1) = \overline{S_1 \Gamma(S_1 B_1)}$.
From Lemma 46, on equation 347, we have $fov(S_1 B_1) \subseteq fov(S_0 \Gamma)$. Obviously $fov(S_0 \Gamma) \subseteq fov(S_1 \Gamma)$, since $S_1 = R' \cdot S_0$ for some R'. So $fov(S_1 B_1) \subseteq fov(S_1 \Gamma)$, namely $ftv(S_1 B_1) - ftv(S_1 \Gamma)$ are all meta type variables.
By lemma 36 $R S_1 B_2 = R S_1(\Gamma_{agen}(B_1)) = R S_1 \overline{S_1 \Gamma(S_1 B_1)} <: \overline{R S_1 S_1 \Gamma(R S_1 S_1 B_1)} = \overline{R S_1 \Gamma(R S_1 B_1)} = \overline{S S_0 \Gamma(R S_1 B_1)}$.
And it remains to prove $\overline{S S_0 \Gamma(R S_1 B_1)} <: S S_0 \Gamma_{gen}(A_1)$. It can be easily derived from equation 349.

The two parts of lemma rely on each other. This lemma holds because part 1 calls part 2 with a smaller height and part 2 calls part 1 with the same height. So it will terminate. □

Theorem 4 (Completeness). *If $\Gamma \vdash e \Rightarrow A$, then for a fresh name supply N_0, that $([], N_0) \shortmid \Gamma \vdash e \Rightarrow B \hookrightarrow (S_1, N_1)$, and $\exists R$, that $\overline{\Gamma(R S_1 B)} <: \overline{\Gamma(A)}$.*

Follows directly by lemma 50. □

E.4 Extension of Pairs

This section talks about how to extend the algorithmic type system with pairs. To do so, there are several changes in the formalizations, and more cases after the induction in proofs.

Unification The unification rules for pairs are given in Figure 17, which are quite standard.

$\boxed{S_0 \vdash \tau_1 = \tau_2 \hookrightarrow S_1}$
τ_1, τ_2 inputs

$$\frac{S_0 \vdash \tau_1 = \tau_1' \hookrightarrow S_1 \qquad S_1 \vdash \tau_2 = \tau_2' \hookrightarrow S_2}{S_0 \vdash (\tau_1, \tau_2) = (\tau_1', \tau_2') \hookrightarrow S_2}\ \text{AU-PAIR}$$

$$\frac{S_0 \vdash \tau_1 = \tau_1' \hookrightarrow S_1}{S_0 \vdash \mathbf{fst}\,\tau_1 = \mathbf{fst}\,\tau_1' \hookrightarrow S_1}\ \text{AU-FST} \qquad \frac{S_0 \vdash \tau_1 = \tau_1' \hookrightarrow S_1}{S_0 \vdash \mathbf{snd}\,\tau_1 = \mathbf{snd}\,\tau_1' \hookrightarrow S_1}\ \text{AU-SND}$$

Fig. 17. Unification extends Figure 13 with pairs.

Soundness of Unification. Following proof extends the proof for Lemma 37.

- Case AU-PAIR. Similar as AU-Fun.
- Case AU-FST. Follows directly from the hypothesis.
- Case AU-SND. Similar as AU-FST.

□

Completeness of Unification. Following proof extend the proof for Lemma 47.

- τ_1, τ_2 are both pair types. So $\tau_1 = (\tau_{11}, \tau_{12})$, $\tau_2 = (\tau_{21}, \tau_{22})$. By inversion we have

$$SS_0\tau_{11} = SS_0\tau_{21} \tag{350}$$
$$SS_0\tau_{12} = SS_0\tau_{22} \tag{351}$$

Since

$$\begin{aligned} &\langle |range(S_0) \cup ftv(\tau_{11}, \tau_{21})|, size(\tau_{11}, \tau_{21})\rangle \\ &< \langle |range(S_0) \cup ftv((\tau_{11}, \tau_{12}), (\tau_{21}, \tau_{22}))|, size((\tau_{11}, \tau_{12}), (\tau_{21}, \tau_{22}))\rangle \end{aligned} \tag{352}$$

By induction on equation 350, we have

$$S_0 \vdash \tau_{11} = \tau_{21} \hookrightarrow S_1 \tag{353}$$
$$SS_0 = R_1S_1 \tag{354}$$

Moreover, $vars(R_1) \subseteq vars(S) \cup vars(S_0) \cup ftv(\tau_{11}, \tau_{21})$.
And by soundness of unification (lemma 37) on equation 353, we have $vars(S_1) \subseteq vars(S_0) \cup ftv(\tau_{11}, \tau_{21})$.
From $SS_0\tau_{12} = SS_0\tau_{22}$, with equation 354 we have $R_1S_1\tau_{12} = R_1S_1\tau_{22}$.
By lemma 42 that $range(S_1) \subseteq range(S_0) \cup ftv(\tau_{11}, \tau_{21})$. So we have

$$\begin{aligned} &\langle |range(S_1) \cup ftv(\tau_{12}, \tau_{22})|, size(\tau_{12}, \tau_{22})\rangle \\ &< \langle |range(S_0) \cup ftv((\tau_{11}, \tau_{12}), (\tau_{21}, \tau_{22}))|, size((\tau_{11}, \tau_{12}), (\tau_{21}, \tau_{22}))\rangle \end{aligned} \tag{355}$$

then by induction, we have

$$S_1 \vdash \tau_{12} = \tau_{22} \hookrightarrow S_2 \tag{356}$$
$$R_1S_1 = R_2S_2 \tag{357}$$

And $vars(R_2) \subseteq vars(R_1) \cup vars(S_1) \cup ftv(\tau_{12}, \tau_{22})$.
So, feed S_2 and R_2 to the goal, we have $S_1 \vdash \tau_1 = \tau_2 \hookrightarrow S_2$ by using AU-Pair with equation 353 and 356. And $SS_0 = R_2 S_2$ by transitivity with equation 354 and 357.

□

$\boxed{(S_0, N_0) \vdash^{()} A = (A_1, A_2) \hookrightarrow (S_1, N_1)}$
A input, A_1, A_2 outputs

$$\frac{S_0 \vdash \widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1}{(S_0, N_0\widehat{\alpha}_1\widehat{\alpha}_2) \vdash^{()} \widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow (S_1, N_0)}\ \text{AP-Mono}$$

$$\frac{}{(S_0, N_0) \vdash^{()} (A, B) = (A, B) \hookrightarrow (S_0, N_0)}\ \text{AP-Pair}$$

Fig. 18. Pair Unification

Pair Unification Similar as Arrow Unification, we need a Pair Unification process for pairs, as given in Figure 18.

Lemma 51 (Soundness of Pair Unification). *if* $(S_0, N_0) \vdash^{()} A = (A_1, A_2) \hookrightarrow (S_1, N_1)$*, then* $S_1 A = (S_1 A_1, S_1 A_2)$*, and* $\exists R$*, that* $S_1 = R \cdot S_0$*. Moreover,* $ftv((A_1, A_2))$*, and* $vars(S_1)$ *are all subsets of* $vars(S_0) \cup ftv(A) \cup (N_0 - N_1)$*;* $vars(R) \subseteq ftv(S_0 A) \cup (N_0 - N_1)$*.*

Proof. By induction on pair unification relation. We analyze each case.

- Case AP-MONO. $S_1\widehat{\alpha}_1 = (S_1\widehat{\alpha}_1, S_1\widehat{\alpha}_2)$ comes directly by the soundness of unification (lemma 37), with the same R. $ftv((\widehat{\alpha}_1, \widehat{\alpha}_2)) = \{\widehat{\alpha}_1, \widehat{\alpha}_2\} = (N_0\widehat{\alpha}_1\widehat{\alpha}_2) - N_0$. So $ftv((\widehat{\alpha}_1, \widehat{\alpha}_2)) \subseteq (N_0\widehat{\alpha}_1\widehat{\alpha}_2) - N_0$. And from soundness of unification, we get $vars(R) \subseteq ftv(S_0\widehat{\alpha}, S_0(\widehat{\alpha}_1, \widehat{\alpha}_2))$. Therefore $vars(R) \subseteq ftv(S_0\widehat{\alpha}) \cup ftv((\widehat{\alpha}_1, \widehat{\alpha}_2)) \subseteq ftv(S_0\widehat{\alpha}) \cup ftv(N_0\widehat{\alpha}_1\widehat{\alpha}_2 - N_0)$.
  Finally, from the soundness of unification, we know $S_1 = R \cdot S_0$. So $vars(S_1) \subseteq vars(R) \cup vars(S_0) \subseteq vars(S_0) \cup ftv(\widehat{\alpha}) \cup ftv(N_0\widehat{\alpha}_1\widehat{\alpha}_2 - N_0)$. The case is finished.
- Case AP-Pair. Choose R to be empty, each goal holds trivially.

□

Lemma 52 (Completeness of Pair Unification). *If* $SS_0 A = (B, C)$*, then for any fresh name supply* N_0*, we have for some* S_1*,* $(S_0, N_0) \vdash^{()} A = (A_1, A_2) \hookrightarrow (S_1, N_1)$*, and* $\exists R$*, that* $S \cdot S_0 = R \cdot S_1\backslash_{N_0 - N_1}$*, and* $RS_1A_1 = B$*,* $RS_1A_2 = C$*. Moreover,* $vars(R) \subseteq vars(S) \cup vars(S_0) \cup ftv(A) \cup (N_0 - N_1)$*.*

Proof. By case analysis on the type structure of A.

- A is Int or an ordinary variable or function. Impossible.
- A is a meta variable $\widehat{\alpha}$. Then both B and C are monotypes. Then we would like to use rule AP-Mono, where we have $(S_0, N_0\widehat{\alpha}_1\widehat{\alpha}_2) \vdash^{()} \widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow (S_1, N_0)$, if given $S_0 \vdash (\widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2)) \hookrightarrow S_1$. So our goal now is to prove the later one. There are two sub-cases:
  - $\widehat{\alpha} \notin dom(S_0)$. Here we are going to use AU-Var1, because definitely $\widehat{\alpha} \notin ftv(S_0(\widehat{\alpha}_1, \widehat{\alpha}_2)) = \{\widehat{\alpha}_1, \widehat{\alpha}_2\}$ since $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ come from a fresh name supply. So $S_1 = [\![\widehat{\alpha} \mapsto (\widehat{\alpha}_1, \widehat{\alpha}_2)]\!] \cdot S_0$.
    Then in order to prove $S \cdot S_0 = R \cdot S_1 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\alpha} \mapsto (\widehat{\alpha}_1, \widehat{\alpha}_2)]\!] \cdot S_0 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, pick $R = S \cdot [\![\widehat{\alpha}_1 \mapsto B]\!] \cdot [\![\widehat{\alpha}_2 \mapsto C]\!]$.
  - $\widehat{\alpha} \in dom(S_0)$. Then we case analyze the form of $S_0\widehat{\alpha}$
    - $S_0\widehat{\alpha}$ is Int or an ordinary variable or a function. Impossible.
    - $S_0\widehat{\alpha} = \widehat{\beta}$ where $\widehat{\beta} \neq \widehat{\alpha}$. Also because S_0 is well-defined, so $\widehat{\beta} \notin dom(S_0)$. Then in order to give $S_0 \vdash \widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1$, we are going to use rule AU-BVar1 and turn to give $S_0 \vdash S_0\widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1$, namely $S_0 \vdash \widehat{\beta} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1$. For this one, we are going to use rule AU-Var1 to conclude $S_1 = [\![\widehat{\beta} \mapsto (\widehat{\alpha}_1, \widehat{\alpha}_2)]\!] S_0$.
      In order to prove $S \cdot S_0 = R \cdot S_1 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\beta} \mapsto (\widehat{\alpha}_1, \widehat{\alpha}_2)]\!] \cdot S_0 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_1}$, we pick $R = S \cdot [\![\widehat{\alpha}_1 \mapsto B]\!] \cdot [\![\widehat{\alpha}_2 \mapsto C]\!]$.
    - $S_0\widehat{\alpha} = (D_1, D_2)$. Then in order to give $S_0 \vdash \widehat{\alpha} = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1$, we are going to use rule AU-BVar1 and turn to give $S_0 \vdash (D_1, D_2) = (\widehat{\alpha}_1, \widehat{\alpha}_2) \hookrightarrow S_1$. We assume D_1 and D_2 are not meta type variables, because the cases when they are meta variables are similar. Then we can apply rule AU-Pair and with a result $S_1 = [\![\widehat{\alpha}_2 \mapsto D_2]\!] \cdot [\![\widehat{\alpha}_1 \mapsto D_1]\!] \cdot S_0$.
      In order to prove $S \cdot S_0 = R \cdot S_1 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, namely $S \cdot S_0 = R \cdot [\![\widehat{\alpha}_2 \mapsto D_2]\!] \cdot [\![\widehat{\alpha}_1 \mapsto D_1]\!] \cdot S_0 \backslash_{\widehat{\alpha}_1, \widehat{\alpha}_2}$, we choose $R = S$. It is easy to derive this holds so we are done.
- A is a pair type so $A = (D_1, D_2)$. So $SS_0D_1 = B$ and $SS_0D_2 = C$. Then by AF-Pair we have $(S_0, N_0) \vdash^{()} (D_1, D_2) = (D_1, D_2) \hookrightarrow (S_0, N_0)$, namely $A_1 = D_1$ and $A_2 = D_2$. Then choose $R = S$ and we are done.

□

Subtyping The algorithmic subtyping rules for pairs are given in Figure 19, which make use of the Pair Unification process.

Subtyping Subsitution. Following proof extends the proof for the Part 1 of Lemma 34.

- Case S-Pair. By induction hypothesis, we have $S_1A_1 <: S_1B_1$ and $S_1A_2 <: S_1B_2$. Follows directly by using rule S-Pair.

□

Soundness of Subtyping. Following proof extends the proof for Lemma 39.

- Case AS-PairL. Similar as AS-FunL.
- Case AS-PairR. Similar as AS-FunR.

□

$\boxed{(S_0, N_0) \vdash A \ <: \ B \hookrightarrow (S_1, N_1)}$
A, B inputs

$$\frac{(S_0, N_0) \vdash^{()} A = (A_1, A_2) \hookrightarrow (S_1, N_1) \quad (S_1, N_1) \vdash A_1 \ <: \ B \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash A_2 \ <: \ C \hookrightarrow (S_3, N_3)}{(S_0, N_0) \vdash A \ <: \ (B, C) \hookrightarrow (S_3, N_3)} \text{AS-PairR}$$

$$\frac{(S_0, N_0) \vdash^{()} A = (A_1, A_2) \hookrightarrow (S_1, N_1) \quad (S_1, N_1) \vdash B \ <: \ A_1 \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash C \ <: \ A_2 \hookrightarrow (S_3, N_3)}{(S_0, N_0) \vdash (B, C) \ <: \ A \hookrightarrow (S_3, N_3)} \text{AS-PairL}$$

Fig. 19. Subtyping extends Figure 15 with pairs.

Completeness of Subtyping. Following proof extends the proof for Lemma 49.

– Case S-Pair. Similar as S-Fun.

□

$\boxed{(S_0, N_0) \mid \Gamma \mid \Psi \vdash e \ \Rightarrow \ A \hookrightarrow (S_1, N_1)}$
Γ, Ψ, e inputs, A output

$$\frac{(S_0, N_0) \mid \Gamma \vdash e_1 \ \Rightarrow \ A_1 \hookrightarrow (S_1, N_1) \qquad (S_1, N_1) \mid \Gamma \vdash e_2 \ \Rightarrow \ A_2 \hookrightarrow (S_2, N_2)}{(S_0, N_0) \mid \Gamma \vdash (e_1, e_2) \ \Rightarrow \ (A_1, A_2) \hookrightarrow (S_2, N_2)} \text{AT-Pair}$$

$$\frac{(S_0, N_0) \mid \Psi \vdash \forall ab.(a, b) \to a \ <: \ A \hookrightarrow (S_1, N_1)}{(S_0, N_0) \mid \Gamma \mid \Psi \vdash \mathbf{fst} \ \Rightarrow \ A \hookrightarrow (S_2, N_2)} \text{AT-Fst}$$

$$\frac{(S_0, N_0) \mid \Psi \vdash \forall ab.(a, b) \to b \ <: \ A \hookrightarrow (S_1, N_1)}{(S_0, N_0) \mid \Gamma \mid \Psi \vdash \mathbf{snd} \ \Rightarrow \ A \hookrightarrow (S_2, N_2)} \text{AT-Snd}$$

Fig. 20. Typing extends Figure 16 with pairs.

Typing The algorithmic typing rules for pairs are given in Figure 20.

Typing Subsitution. Following proof extends the proof for Part 1 of Lemma 35.

– Case AT-Pair. By induction we have $S_1\Gamma \mid S_1\Psi \vdash e_1 \ \Rightarrow \ S_1 A$, and $S_1\Gamma \mid S_1\Psi \vdash e_2 \ \Rightarrow \ S_1 B$, follows directly by AT-Pair.
– Case AT-Fst. Follows directly by subtyping substitution (Lemma 34).
– Case AT-Snd. Similar as AT-Fst.

□

Soundness of Typing. Following proof extends the proof for Lemma 41.

– Case AT-Pair. Given

$$(S_0, N_0) \vdash \Gamma \vdash e_1 \;\Rightarrow\; A_1 \hookrightarrow (S_1, N_1) \tag{358}$$
$$(S_1, N_1) \vdash \Gamma \vdash e_2 \;\Rightarrow\; A_2 \hookrightarrow (S_2, N_2) \tag{359}$$

By induction hypothesis, we have

$$S_1\Gamma \vdash e_1 \;\Rightarrow\; S_1 A_1 \tag{360}$$
$$S_1 = R_1 \cdot S_0 \tag{361}$$
$$S_2\Gamma \vdash e_2 \;\Rightarrow\; S_2 A_2 \tag{362}$$
$$S_2 = R_2 \cdot S_1 \tag{363}$$

Moreover, both $vars(S_1)$ and $vars(A_1)$ are subsets of $vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_0 - N_1)$, $vars(R_1) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_0 - N_1)$.
And, both $vars(S_2)$ and $vars(A_2)$ are subsets of $vars(S_1) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_1 - N_2)$, $vars(R_2) \subseteq ftv(S_1\Gamma) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$.
By applying the first part of typing substitution (Lemma 35) on equation 360 with R_2, we have

$$R_2S_1\Gamma \vdash e_1 \;\Rightarrow\; R_2S_1A_1 \tag{364}$$

namely

$$S_2\Gamma \vdash e_1 \;\Rightarrow\; S_2A_1 \tag{365}$$

So by rule T-Pair and equation 362 and 365 we have

$$S_2\Gamma \vdash (e_1, e_2) \;\Rightarrow\; S_2(A_1, A_2) \tag{366}$$

By equation 361 and 363, we have

$$S_2 = R_2 \cdot R_1 \cdot S_0 \tag{367}$$

Moreover, both $vars(S_2)$ and $vars((A_1, A_2))$ are subsets of $vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_0 - N_1) \cup (N_1 - N_2)$, namely $vars(S_0) \cup ftv(\Gamma) \cup ftv(\Psi) \cup (N_0 - N_2)$.
And $vars(R_2 \cdot R_1) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_0 - N_1) \cup ftv(S_1\Gamma) \cup ftv(S_1\Psi) \cup (N_1 - N_2)$. Because we know that $ftv(S_1\Gamma) = ftv(R_1S_0\Gamma) \subseteq vars(R_1) \cup ftv(S_0\Gamma)$, similar for $ftv(S_1\Psi)$, so we can get $vars(R_2 \cdot R_1) \subseteq ftv(S_0\Gamma) \cup ftv(S_0\Psi) \cup (N_0 - N_2)$ and we are done.
– Case AT-Fst. Follows directly by soundness of subtyping (Lemma 39), by noticing $ftv(\forall ab.(a, b) \to a) = \emptyset$.
– Case AT-Snd. Similar as AT-Fst.

□

Completeness of Typing. Following proof extends the proof for Part 1 of Lemma 50.

– Cast AT-Pair. We have $\Psi = \varnothing$, $A = (A_1, A_2)$, and $e = (e_1, e_2)$, and

$$SS_0\Gamma \vdash e_1 \;\Rightarrow\; A_1 \tag{368}$$
$$SS_0\Gamma \vdash e_2 \;\Rightarrow\; A_2 \tag{369}$$

By induction hypothesis on equation 368 we have for some B_1, N_1, S_1, R_1,

$$(S_0, N_0) \vdash \Gamma \vdash e_1 \;\Rightarrow\; B_1 \hookrightarrow (S_1, N_1) \tag{370}$$
$$S \cdot S_0 = R_1 \cdot S_1 \backslash_{N_0 - N_1} \tag{371}$$
$$\overline{SS_0\Gamma(R_1S_1B_1)} \;<:\; \overline{SS_0\Gamma(A_1)} \tag{372}$$

Moreover, N_1 is fresh.
We substitute equation 371 on 369 we get

$$R_1S_1\Gamma \vdash e_2 \;\Rightarrow\; A_2 \tag{373}$$

By induction hypothesis, we have for some B_2, N_2, S_2, R_2

$$(S_1, N_1) \vdash \Gamma \vdash e_2 \;\Rightarrow\; B_2 \hookrightarrow (S_2, N_2) \tag{374}$$
$$R_1 \cdot S_1 = R_2 \cdot S_2 \backslash_{N_1 - N_2} \tag{375}$$
$$\overline{R_1S_1\Gamma(R_2S_2B_2)} \;<:\; \overline{R_1S_1\Gamma(A_2)} \tag{376}$$

Moreover, N_2 is fresh.
By rule AT-Pair and by equation 370 and 374 we have

$$(S_0, N_0) \vdash \Gamma \vdash (e_1, e_2) \;\Rightarrow\; (B_1, B_2) \hookrightarrow (S_2, N_2) \tag{377}$$

Notice $SS_0\Gamma = R_1S_1\Gamma = R_2S_2\Gamma$.
Consider $\bar{c} = ftv(R_2S_2B_2) - ftv(R_1S_1\Gamma)$. It must be $\bar{c} \not\subseteq dom(R_2S_2)$. Therefore $\bar{c} \not\subseteq dom(R_1S_1)$. Also $\bar{c} \not\subseteq ftv(\Gamma)$. Furthermore, for every $c \in \bar{c}$, either $c \in ftv(B_2)$, or there exists d, $d \in ftv(B_2)$, $d \in dom(R_2S_2)$, and $c \in ftv(R_2S_2B_2)$.

- $c \in ftv(B_2)$. By soundness of typing on equation 374, $ftv(B_2) \subseteq vars(S_1) \cup ftv(\Gamma) \cup N_1 - N_2$. We know $c \notin ftv(\Gamma)$. If c comes from $vars(S_1)$, it means $c \in range(S_1)$. However in this case c cannot be introduced into B directly. So there must be a d, that $c \in ftv(S_1 d)$, and $d \in ftv(\Gamma)$. But then $c \in ftv(R_1S_1\Gamma)$ a contradiction. So the only case remains is $c \in N_1 - N_2$. In this case, c cannot in $vars(R_1S_1)$. We use V to represent the substitution that maps c to a fresh variable c_1. So, we have

$$R_1 \cdot S_1 = (V \cdot R_2) \cdot S_2 \backslash_{N_1 - N_2} \tag{378}$$
$$\overline{R_1S_1\Gamma((VR_2)S_2B_2)} \;<:\; \overline{R_1S_1\Gamma(A_2)} \tag{379}$$

From equation 371 and 378 we can get

$$S \cdot S_0 = (VR_2) \cdot S_2 \backslash_{N_0 - N_2} \tag{380}$$

And from equation 372 and 379, and we know $\overline{c_1} = ftv((VR_2)S_2B_1) - ftv(\Gamma)$ are fresh to the free type variables in $ftv((VR_2)S_2B_1)$, so

$$\overline{SS_0\Gamma((VR_2)S_2B_1, (VR_2)S_2B_2)} \ <: \ \overline{SS_0\Gamma((A_1, A_2))} \tag{381}$$

So we are done.

- $d \in ftv(B_2)$, $d \in dom(R_2S_2)$, $c \in ftv(R_2S_2d)$. Again by soundness of typing, $d \in vars(S_1) \cup ftv(\Gamma) \cup N_1 - N_2$. But if $d \in ftv(\Gamma)$, then $c \in ftv(R_2S_2\Gamma)$, leads to a contradiction. If $d \in vars(S_1)$, again it can only be introduced by a $d_1 \in ftv(\Gamma)$, leads to a contradiction. So $d \in N_1 - N_2$. In this case, d cannot in $vars(R_1S_1)$.
  Since $d \notin vars(S_1)$, so it cannot in $dom(S_2)$, so it is in $dom(R_2)$.
  We use V to represent substitution after the changes in R_2 that substitute the type variable c in the solution of d with some fresh variable c_1.

$$R_1 \cdot S_1 = V \cdot S_2 \backslash_{N_1 - N_2} \tag{382}$$

$$\overline{R_1S_1\Gamma(VS_2B_2)} \ <: \ \overline{R_1S_1\Gamma(A_2)} \tag{383}$$

From equation 371 and 382 we can get

$$S \cdot S_0 = V \cdot S_2 \backslash_{N_0 - N_2} \tag{384}$$

Substituting equation 375 and 383, we can get

$$\overline{SS_0\Gamma(VS_2B_1, VS_2B_2)} \ <: \ \overline{SS_0\Gamma((A_1, A_2))} \tag{385}$$

So we are done.

– Case AT-Fst. Follows directly by completeness of subtyping.
– Case AT-Snd. Similar as AT-Fst.